Reo: Enhancing Reliability and Efficiency of Object-based Flash Caching

Jian Liu
Computer Science & Engineering
Louisiana State University
jliu@csc.lsu.edu

Kefei Wang
Computer Science & Engineering
Louisiana State University
kwang@csc.lsu.edu

Feng Chen
Computer Science & Engineering
Louisiana State University
fchen@csc.lsu.edu

***Abstract*—The fast-pace advancement of flash technology has enabled us to build a very large-capacity cache system at a low cost. However, the reliability of flash devices still remains a non-negligible concern, especially for flash-based cache. This is for two reasons. First, corruption of dirty data in cache would cause a permanent loss of user data. Second, warming up a huge-capacity cache would take an excessively long period of time. In this paper, we present a highly reliable, efficient, object-based flash cache, called *Reo*. Reo is designed to leverage the highly expressible object interface to exploit the rich semantic knowledge of the flash cache manager. Reo has two key mechanisms, differentiated data redundancy and differentiated data recovery, to make a flash cache highly reliable, and in the meantime, still remains space efficient for high cache hit ratio. We have prototyped Reo based on *open-osd*, an open-source implementation of T10 Object Storage Device (OSD) in Linux. Our experimental results show that compared to uniform data protection, Reo achieves graceful performance degradation and prioritized recovery upon device failures. Compared to full replication, Reo is more space efficient and delivers up to 3.1 times of the cache hit ratio and up to 3.6 times of the bandwidth.**

I. Introduction

In today's data centers, flash-based cache systems are being widely deployed for supporting high-speed data services [1], [2], [3]. The recent technical breakthroughs in flash technologies, such as 3D Vertical NAND (V-NAND) and Quad-Level Cell (QLC), have dramatically increased the device capacity and reduced the cost, which enables us to build a very large, high-speed flash cache for fast data accesses.

However, a long-existing concern that still remains in the industry is the *reliability* of flash memory. As a type of EEPROM devices, the current NAND flash technology has a fundamental limitation—a flash memory cell wears out after a limited number (1,000-5,000) of Program/Erase (P/E) cycles, eventually leading to various hardware reliability issues, from partial data loss to a complete device failure [4]. As the bit density continues to increase (e.g., four bits per cell for QLC), this lifetime problem is expected to be further exacerbated rather than improved, meaning that the data reliability issue will become even more challenging in the future.

A. Reliability Challenges in Flash-based Cache

A key application of flash technology is high-speed storage cache. On one hand, the rapid growth of flash capacity enables us to cache an unprecedentedly large volume of data in flash at a very low cost (e.g., 1 Terabyte for less than $150). On the other hand, an unexpected device failure could cause detrimental results, which must be carefully addressed.

The most critical threat is *permanent loss* of user data. In a typical write-back cache, updates to data are temporarily held in cache and asynchronously flushed to the backend data store (e.g., a database or a file system). It improves the write performance. However, if any such "dirty data" in cache are corrupted or inaccessible, it would cause an irrecoverable data loss, leaving an obsolete, inconsistent copy in the data store. Such a cache failure could result in various catastrophic consequences, e.g., system crash, application malfunctioning, service disruption, silent data corruption, etc. Due to the violation of the service agreement with customers, even more profound damages could happen, such as legal dispute, economic penalty, irreparable brand image, not to mention the loss of customers, market share, and investors.

Even for a read-only cache, cache device failures would significantly and abruptly impair the caching services and cause impact to the whole system. In data centers, high-speed caching services are often the first line of defence for various computing services. A cache device failure could render a complete loss of caching services, affecting a large number of clients and enforcing them to directly retrieve data from the backend data store. Consequently, the backend servers would be inevitably overloaded, resulting in intolerably long response time, dropping client requests, and spreading the impact of service degradation across the entire system. More importantly, with today's flash technology, a single-device capacity already grows to Terabyte level. An enterprise flash array can easily accommodate tens of Terabytes of data. Re-warming up the entire cache from scratch again would take an excessively long period of time, rendering the underperformance of caching services for hours to even days [5].

B. Design Goals

In order to address the above-said reliability challenges, we desire to build a highly reliable, large-capacity cache based on high-speed flash SSDs. It is non-trivial to achieve this goal. We must address three critical issues as follows.

• *The cache system needs to provide a graceful degradation upon device failures.* Upon device failures, we desire to see a *gracefully* degraded service quality, rather than an abrupt,

complete loss of caching services. Ideally, the cache system should continue to provide caching services, proportional to the surviving devices and uncorrupted data.

• *The failure recovery is time consuming and its impact to user experience should be minimized*: We desire to accelerate the recovery process, bring up the quality of caching services (i.e., hit ratio) to the normal state as quickly as possible, and minimize the impact of the recovery process to the other incoming caching requests.

• *The limited cache space must be efficiently used.* For recovery, certain level of data redundancy is needed upon device failures. However, such redundant information also occupies flash space, reducing the effectively usable cache space and negatively affecting the cache hit ratio. We need to reach the best balance between data reliability and space efficiency.

C. Reo: A Reliable and Efficient Object-based Flash Cache

In this paper, we present a *R*eliable, *E*fficient, *O*bject-based flash cache system, called *Reo*. Reo is built on an array of high-speed flash SSDs. It is designed to handle multiple partial or complete device failures, providing highly reliable, continuous caching services. The essential goal is to retain the caching services online as much as possible, even under harsh situations, such as multi-device failures.

A key design principle of Reo is—*differentiate data objects based on their semantic importance and handle them differently*. Such an object-based data differentiation enables us to provide different levels of data redundancy and to prioritize the recovery process to handle device failures.

Leveraging the expressive and flexible object storage interface, Reo exploits applications' semantic knowledge about data objects and manages the cache data in a highly efficient and reliable manner, through two key mechanisms:

• **Differentiated Redundancy**. To maximize data reliability and also the efficiency of cache space usage, Reo assigns different data redundancy levels according to the semantic importance of data objects. In particular, dirty and hot data objects are given a high redundancy level to survive many device failures, while a low redundancy level is given to low-priority objects, such as cold and clean objects, saving huge cache space.

Such a fine-grained, differentiated data redundancy brings two benefits. First, it is highly space efficient. Owning to the workloads' locality, most data objects are relatively cold and clean and do not need to be maintained in a high redundancy level. The saved flash space can be effectively used for caching, improving the hit ratio. Second, it enables a graceful degradation of caching services. Due to the multiple levels of data redundancy, losing a subset of flash devices or data in cache only results in a partial loss of the relatively less important data (i.e., the clean and cold data), making the caching services remain available to its maximum extent.

• **Differentiated Recovery**. Upon device failures, the surviving devices still contain partially usable or repairable data objects. If an object is immediately usable, Reo leverages the flexible object interface to locate its position and retrieve the data; if the object is not directly accessible but can be reconstructed, Reo performs the recovery process to restore the on-demand data first. To further mitigate the long latency of reconstructing data objects, Reo proactively rebuilds the data objects in advance, according to the objects' semantic importance. In particular, hot data objects are reconstructed first, since it is more likely to be requested soon, while cold objects are reconstructed at a lower priority. By differentiating the semantic importance of data objects, we can quickly bring the caching services back to its maximum potential.

With differentiated redundancy and differentiated recovery, Reo is able to optimize the flash space usage to maximize the protection for important data objects while minimizing the demanded space and recovery time. We have built a prototype of Reo based on *open-osd* [6], which is an open-source implementation of T10 Object Storage Device (OSD) in Linux. Our experimental results show that compared to uniform data protection, Reo achieves graceful performance degradation and prioritized recovery upon device failures. Compared to full replication, Reo is more space efficient and delivers up to 3.1 times of the cache hit ratio and up to 3.6 times of the bandwidth.

The rest of this paper is organized as follows. Section II gives the background. Section III discusses the related work. Section IV describes the design. Section V introduces the implementation. Section VI presents the evaluation results. The final section concludes this paper.

II. Background

A. Object-based Storage

Object-based Storage Device (OSD) [7] is an extension to T10 Small Computer System Interface (SCSI) command set [8]. It was initially proposed to overcome the limitations of the narrow block-based interface via a more expressive object-based interface [9]. As a self-management data storage, object-based storage offloads its data storage component to the OSD from the original file system, as shown in Figure 1. More importantly, the object interface allows applications to deliver abundant semantic information from the application level to the OSD device level, enabling the OSD device to make an intelligent and application-aware data management.

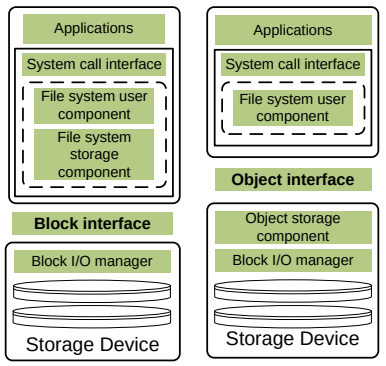


Fig. 1: Blocked-based storage vs. object-based storage [9].

TABLE I: An illustration of different types of objects in OSD.

Type	PID	OID	Description
Root Object	0x0	0x0	Recording the global information of the OSD
Partition Object	0x10000 and above	0x0	Recording the set of collection and user objects within the partition
Collection Object	0x10000 and above	0x10000 and above	A logical collection of user objects for fast indexing
User Object	0x10000 and above	0x10000 and above	A regular user data (e.g., a file or directory)
Super Block Object	0x10000	0x10000	Recording the super block information
Device Table Object	0x10000	0x10001	Recording the device information
Root Directory Object	0x10000	0x10002	Recording the root directory / information

Note: The OSD-2 documentation [7] defines four basic types of objects, Root, Partition, Collection, and User. The implementation of exofs [10] in Linux reserves OID# 0x10000-0x10002 for Super Block, Device Table, and Root Directory objects as metadata.

As shown in Figure 2, OSD manages the storage data in the form of *objects*. Each object has an exclusive *partition ID* (PID) and an *object ID* (OID), which is the only identifier for the object within the device. In general, there are four different types of data objects in the storage. For each OSD logical unit, there is a *root* object, whose PID and OID are both set to 0x0. The root object records the global information about the OSD, such as the device capacity, the number of partitions. Another important type of object is *partition* object. To facilitate the management of data objects, the whole OSD logical unit is divided into multiple *partitions*. Each partition contains a set of *collection* objects and *user* objects. The collection objects are created for the sake of fast indexing and grouping. A user object belongs to no or multiple collections [7]. All the collection objects and user objects within one partition share the same PID but have distinct OIDs.

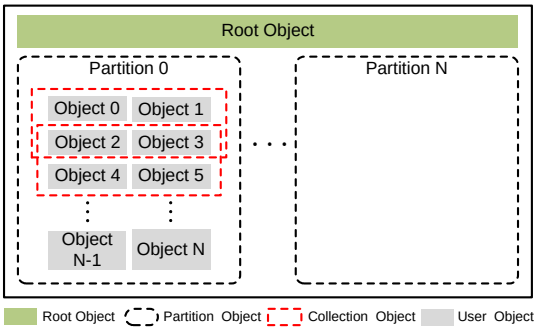


Fig. 2: Partition, Collection, and User objects in OSD.

OSD can be emulated by using iSCSI protocol coupled with the current block-based devices. The *initiator* refers to the host-side of an iSCSI session, while the *target* refers to the server-side of an iSCSI session. On the initiator side, a special file system *exofs*, which has been adopted in the Linux kernel, runs on top of the initiator and exposes a file system interface to the upper-level applications. All the file system metadata (e.g., superblock, inode), regular files, and directories are stored in the OSD in the form of user objects. On the *target* side, it can use a regular file system (e.g., Ext4) to manage the data objects, and a regular database (e.g., SQLite [11]) to manage the related OSD metadata. Table I shows the different types of OSD objects defined in exofs.

B. Data Reliability

For data reliability, certain level of data redundancy needs to be introduced into the storage for recovery upon device failures. There are two typical ways to achieve this purpose.

Data replication: As a simple way to enhance data reliability, multiple replicas can be stored in a faulty-isolated manner (e.g., spread out to separate disks, data nodes, racks, or even data centers). The limitation of data replication is the high storage overhead. For example, to survive M failures, we need to store M extra copies at the cost of storage space. The *storage efficiency* is as low as $\frac{1}{M+1}$ of that without replication.

Therefore, the data replication approach is often applied with a low redundancy level or for small-sized data only [12]. For example, Google File System [13] uses a so-called *triplication* to store three replicas in multiple locations to enhance the reliability and availability, significantly amplifying the storage space cost. Recently, space-efficient erasure coding methods are replacing the triplication method in many distributed systems to reduce the storage overhead [14].

Erasure coding: *Erasure coding* is widely used in data protection and signal processing [12], [14], [15], [16]. The basic idea of erasure coding is to slice data objects into m equal-size data chunks and encode them with k parity chunks into n fragments, and the value of n is the sum of m and k. If the number of corrupted chunks is less than or equal to k, the original data can be recovered using any k surviving fragments, either data chunks or parity chunks. The encoding ratio $\frac{m}{n}$ describes the space efficiency and the redundancy level [15]. Clearly, the smaller an encoding ratio is, the higher reliability can be achieved with the lower space efficiency.

In practice, Reed-Solomon code (RS-code) [17] is a commonly used erasure coding method, which encodes m data chunks into n fragments using a Vandermonde matrix [12], [16]. Since any data update causes a recalculation of the parity, it would incur a large amount of extra disk reads to fetch the involved data chunks, which is called *direct parity-updating*. To alleviate the write amplification problem, another method is called *delta parity-updating* [12]. Upon any update to the current data chunk, the delta between the updated data chunk and the original data is obtained. Using the delta and the original parity can calculate the new updated parity. The relative performance of the two methods depends on the number of involved disk reads. In this paper, we choose the encoding method that incurs the least disk reads.

III. Related Work

Regarding the cost and performance, flash memory falls nicely in the middle of DRAM and hard disk drives, which makes it an ideal storage media to form a new caching layer between the two. A large body of research has studied the design of flash-based cache in prior work (e.g., [1], [2], [3], [18], [19], [20], [21], [22], [23], [24]). Although the increase of bit density has significantly improved the capacity and cost efficiency, the device reliability becomes an important concern [4]. Our prior study [25] briefly outlines the key reliability issues in flash-based caches.

For a large-capacity flash cache, how to quickly warm up the cache and provide effective caching services is a new challenge. Zhang et al. have studied the cache warm-up issue in their early studies [5]. Their solution is to monitor the storage server workload and keep track of important warm data. By proactively preloading the warm data into the cache, the warm-up process can be accelerated, thereby improving the caching performance. In our study, we focus on retaining the most important data as much as possible by creating more chances to allow such data to survive device failures, rather than warming up an empty cache. These two approaches are orthogonal and complementary to each other.

In order to provide differentiated data redundancy and differentiated data recovery, we need to deliver the semantic knowledge from applications to storage devices. Traditional block-based storage only sees a sequence of logical blocks, and the critical semantic information is curtailed by the narrow Logical Block Address (LBA) interface [26]. A variety of prior studies have proposed solutions to close this semantic gap. The "gray-box" [26] approach aims to infer the high-level information (e.g., file system metadata vs. data) without changing the interface. A more aggressive approach is to directly replace the block interface with an object-based interface [9], which is more expressive for the communication between upper-level applications and low-level storage devices. Semantic information can be easily passed to the device level, allowing to offload data management to the storage device level, which understands both hardware characteristics and also application semantics. However, this solution demands a change to the interface and applications. More details about object storage can be found in prior research works [27], [28], [29], [30], [31]. More recently, Mesnier et al. proposed to adopt a moderate change to the LBA interface by allowing applications to classify the data, and to deliver the classification information through unused bits in the unmodified SCSI commands [32]. Our work is based on the object storage approach.

Our study is also related to prior work on D-GRAID [33], which is proposed to improve the availability of RAID storage by using a method similar to that introduced in semantic smart storage [26]. D-GRAID stores the semantically related blocks (e.g., data blocks of one file) in a fault-isolated manner (e.g., concentrated in one disk), and replicates the file system metadata among multiple devices for better fault tolerance. To avoid clustered device failures, Balakrishnan et al. [34] propose to age SSDs within the array at different rates through unevenly distributed writes and parity reshuffle. Zhang et al. [12] also have implemented a solution to improve the performance of in-memory key-value storage. Their solution is to apply different data redundancy techniques to handle large and small data separately. The more space-efficient coding method is used for large data, while the faster data replication method is applied to small ones for reduced computational overhead. Our work differentiates data based on their importance and focuses on providing high reliability guarantees while still achieving high space efficiency for caching.

IV. Design

In this section, we will first describe the overall architecture of *Reo*, and then the design of the two key mechanisms *differentiated data redundancy* and *differentiated data recovery* for achieving high reliability, efficiency, and performance.

A. Architecture Overview

Reo is a highly reliable, efficient, object-based cache built on an array of flash SSDs. Being carefully designed to survive partial or complete device failures, Reo achieves the desirable graceful degradation and prioritized data recovery. To meet this goal, Reo implements two key mechanisms:

- *Differentiated data redundancy*. Reo is an object-based cache. It leverages the expressive object interface to differentiate and manage the cache data in units of individual objects. Exploiting the semantic hints from the cache system, Reo effectively differentiates data objects in cache (e.g., hot vs. cold, dirty vs. clean). Storing data in a fault-isolated manner on an array of flash devices, Reo assigns different data redundancy levels according to their semantic importance, ensuring that critical data objects can survive multi-device failures and provide graceful degradation.
- *Differentiated data recovery*. In order to minimize the impact of disrupted caching services, Reo is also designed to prioritize the data recovery process based on the data importance. In other words, we recover the data objects in the order of their semantic importance to make sure the most likely-to-be-accessed data objects available at the earliest time. Leveraging the object interface, Reo is also able to ensure the live data objects on the surviving devices still accessible, without being affected by the unrelated device failures.

B. Object Storage Interface

A key task of Reo is to differentiate data objects and handle them with distinct policies. In traditional storage systems, applications communicate with storage devices through the Logical Block Address (LBA) interface, which simplifies the data storage abstraction as a sequence of equal-size blocks. In contrast, Reo is an object-based cache built on Object Storage Device (OSD) [7], which is an extension to the T10 Small Command Set Interface (SCSI) standard [8]. Leveraging the expressibility of the object interface, we assign a label, which is in effect a *semantic hint*, with each data object; when the OSD storage receives a labeled object, the object storage system applies the policy accordingly.

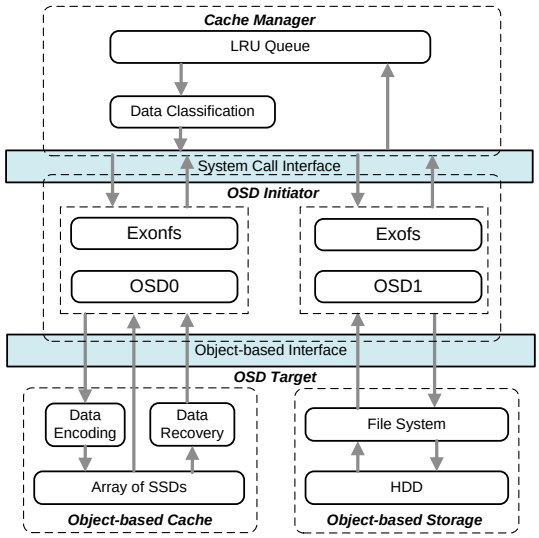


Fig. 3: An illustration of Reo cache architecture.

C. Differentiated Data Redundancy

Data objects in the cache are not equally important—Dirty data objects are more important than clean ones, since they are the only valid copy in the system (not synchronized to the backend data store); the frequently accessed (hot) objects are more important than the infrequently accessed (cold) ones, since these cold data objects are unlikely to be accessed soon. We desire to have the more important data objects remain accessible, even upon cache device failures.

The purpose of differentiated data redundancy is to achieve both *space efficiency* and also *data reliability*, by selectively applying different levels of data redundancy. In particular, a high level of redundancy is applied to "important" data objects, allowing which to survive multi-device failures at the cost of extra space consumption; a low level of redundancy is applied to "unimportant" data objects, which saves space at the cost of being more vulnerable to device failures.

Reo realizes differentiated data redundancy in three steps: (1) classifying data objects in cache, (2) delivering the classification information to the object-based flash cache, and (3) applying the data encoding policy accordingly in the object-based flash cache.

C.1. Data Classification: The cache manager categorizes data objects into different groups, according to their semantic importance; and the object-based flash storage applies the predefined policies, accordingly. This brings three benefits.

First, although various applications may have different types of semantic knowledge, the grouping information can be easily standardized. Second, the storage system can simply apply the corresponding policy without knowing the application details. Third, the interface change can be minimized and easily incorporated in the current object storage interface.

Reo classifies the data in the whole system into four groups, from high to low, according to their importance.

• **Group #0**: *System metadata*. Object storage maintains a set of special objects persistently for organizing the storage data, such as *Root Object*, *Partition Object*, *Super Block Object*, *Device Table Object* and *Root Directory Object*. Besides the standard OSD metadata objects, other application metadata can be also classified into this group. These metadata are crucial to the system integrity and must be given the strongest protection. These metadata objects are given a Class ID "0".

• **Group #1**: *Dirty cache data*. In a write-back cache server, the data updates are temporarily saved in the cache (a.k.a. dirty data). If the dirty data are corrupted, the most recent copy would be permanently lost. Therefore, we need to separate the dirty data objects from the clean ones, offering them a strong protection. Dirty cache data objects are categorized with a Class ID "1".

• **Group #2**: *Hot clean data*. To provide graceful degradation of cache services, we also desire to identify the frequently accessed data objects and ensure they can survive severe device failures, so that we can continue to provide caching services to satisfy most requests. Since most requests still can be hit in the local cache, we will see a graceful degradation of caching services, rather than a sudden, complete service disruption. Reo classifies hot cache data objects with a Class ID "2".

• **Group #3**: *Cold clean data*. The lowest protection is given to cold and clean data objects in the cache. This is for several reasons. First, retaining cold data objects in cache contributes less to the cache hit ratio. Second, a low-level protection means less cache space consumption. The saved space can be used for caching more data. Third, even if a clean data object is unrecoverable, the client still can retrieve it from the backend data store. These data objects are given a Class ID "3".

TABLE II: Classification of data objects in Reo.

Name	Metadata	Read-freq	Dirty	Class ID
A	✓	∼	∼	0
B	✗	∼	✓	1
C	✗	✓	✗	2
D	✗	✗	✗	3

Note: ✓ means that the attribute is true; ✗ means that the attribute is false; ∼ means that the attribute is irrelevant.

Table II summarizes the above-said four data classifications. For the first two classes, we can easily obtain the classification information from the object storage system and the cache manager. We classify the hot and cold clean data objects using a greedy algorithm, as described below.

To identify the hot data, we consider two important factors. (1) *Read Frequency*. The more frequently a data object is accessed in cache, the more important this object is. We associate a counter, *Freq*, with each object to record how many times being accessed since it enters the cache. (2) *Object Size*. The smaller a data object is, the higher priority it has. Given a certain amount of cache space, caching smaller objects can contribute more to improve the cache hit ratio than caching a few large ones. Thus, the priority of a data object is reversely proportional to the object size. Combining both, we calculate an H value, $H = \frac{Freq}{Size}$, for each data object, as an indicator of its *hotness*.

In order to differentiate the hot and cold data objects, we need to determine a proper threshold H_{hot}. An object with an H value greater than H_{hot} is classified as a hot object. We use an adaptive approach to identify the threshold as follows.

We first sort the data objects in the descending order of their H values. Then we presumably add the data objects into the cache one by one, until a predefined data redundancy percentage (e.g., 10%) is reached. That means that all the remaining cache space should only be used to accommodate the cold data objects, which do not need extra space for data redundancy. We record the H value of the last "hot" data object that is included in the cache, as an initial cutoff threshold, H_{hot}. This H_{hot} threshold can be updated periodically to be adaptive to the change of workloads.

C.2. Communication Interface: To enable the communication between the cache manager and the object-based flash storage, we need a mechanism to deliver a variety of management and control information through the object interface. For flexibility and generality, we have designed a special mechanism for such a communication.

We define a special data object (reserved OID `0x10004`) as a communication point. All control messages are encoded into a predefined format and written to this special object. To ensure the object storage truly receives the message, we use `fsync` to bypass the system buffer cache and explicitly flush the write to the storage side immediately. As a message accounts for only a few dozen bytes, the write operation can be completed very quickly. When the object storage receives a write command to this special object, it decodes the message and performs the corresponding operation. Our current prototype defines two types of commands.

• **Classification command**. A classification command delivers a classifier (Class ID) for a specified data object. A classification command has four fields. A header is a string "`#SETID#`", followed by the `PID` and `OID` of the target object, and then the class ID, `CID`. After receiving the command, the object storage sets the class of the object and applies the policy, accordingly.

• **Query command**. A query command retrieves the status of a queried object. A query command consists of six fields. A header is a string "`#QUERY#`", followed by two fields, the object `PID` and `OID`, which specify the target object. Additional fields include the operation type (`R/W`), the offset, and the size of the queried object. If the query is successful, a sense code is returned. For example, the sense code `0x64` indicates that the object storage is full, demanding a cache replacement. A list of the sense codes is shown in Table III.

TABLE III: Sense code definition in Reo.

Sense code	Description
0	The command is successful
-1	The command is unsuccessful
0x63	Data is corrupted
0x64	The cache is full
0x65	Recovery starts
0x66	Recovery ends
0x67	The allocated space for data redundancy is full

C.3. Device Management: The object storage of Reo is built on an array of flash devices, which adopts a stripe-based mechanism similar to Redundant Array of Independent Disks (RAID) [35] to reliably manage data objects in a fault-isolated manner. Figure 4 shows a basic structure, working as follows.

Fig. 4: An illustration of chunks and stripes in Reo.

In the flash array of Reo, the basic management unit is a *stripe*. Each stripe has a unique stripe ID. A stripe is divided into multiple *chunks*, each of which is mapped to a flash device individually. A chunk could be a *data chunk*, which stores the original data object content, or a *parity chunk*, which stores the parity information encoded from the data chunks of the same stripe. We map the parity chunks to the devices in a round-robin manner for an even distribution. We use Reed-Solomon code [17] to encode the parity chunks.

Unlike RAID, a stripe in Reo may contain a variable number of parity chunks. A stripe without a parity chunk contains no redundant information, meaning that if any chunk of this stripe is corrupted, the data chunk is permanently lost; a stripe may contain one or multiple parity chunks, and the number of parity chunks in a stripe determines the redundancy level of the stripe. Clearly, the more parity chunks in a stripe, the more device failures a stripe can survive. A special case is full replication, where the same data chunk is replicated across the entire stripe.

C.4. Data Encoding Policy: The data encoding policy determines how to achieve differentiated data redundancy. The object storage of Reo offers a high data redundancy for data objects with a higher priority. Here we describe how we handle the four distinct data object groups.

• **Group #0: Metadata objects**. System metadata (Class ID: 0) is the most important data in our system. Thus, we apply an aggressive replication policy to offer such objects the strongest protection against device failures. We replicate each metadata object across all the devices, which is similar to how Linux Ext4 file system handles the superblocks. Since metadata objects are small (e.g., the largest one, root directory object, is only 4KB), such a replication is not space-consuming.

• **Group #1: Dirty data objects**. Dirty data objects (Class ID: 1) contain the most recent updates in the system. Since the backend data store is not updated yet, losing a dirty data object would cause a permanent data loss. Thus, we also give them a strong protection by replicating them across the flash devices, similar to the metadata objects. As our aim for read-intensive workloads, the total amount of dirty data objects is small enough to afford such a replication method.

• **Group #2: Hot clean objects**: Hot and clean data objects (Class ID: 2) are frequently accessed. Although losing such data would not cause catastrophic consequences, retrieving them from the backend data store still incurs a long I/O latency. To retain a high hit ratio and ensure a graceful degradation, we desire to offer such hot data objects a reasonable redundancy level to protect them from device failures. Thus, we encode these data objects with two parity blocks in a stripe, which ensures that they can survive no more than two device failures. Due to the cache locality, hot data objects account for a relatively small percentage of the entire cache.
• **Group #3: Cold clean objects**: Cold and clean data objects (Class ID: 3) are the majority in the cache, covering a large portion of the entire data set, which helps eliminate most long-latency accesses to the backend data store. However, relative to the hot objects, these data objects are less frequently accessed, losing them would not cause significant addition of latencies. Thus, we do not provide any data redundancy for such data objects, saving a lot of cache space.

D. Differentiated Data Recovery

In order to accelerate the data recovery process and bring the caching services back up as quickly as possible, Reo adopts a differentiated data recovery mechanism to quickly reconstruct the corrupted data objects.

On-demand access. For a storage failure involving a subset of devices, the other surviving devices still contain a large amount of live data objects, which remain accessible in complete. Reo manages cache data in objects rather than in blocks. Thus, Reo is fully aware if an object is (1) immediately accessible, (2) corrupted but recoverable, or (3) irrecoverable.

Upon an object request, we first check if the object is still available. If the data loss exceeds the recovery capability (greater than k), the object is permanently lost. Otherwise, there are two possibilities. One is that the object itself is still alive and we can directly return the object. The other possible case is that the object is corrupted but can be reconstructed using the remaining data chunks and the parity chunks through the encoding process. In this case, we need to read the surviving chunks, reconstruct the data, and return to satisfy the request.

Data reconstruction. When a spare device is inserted into the flash array, the object storage of Reo initiates a data reconstruction process. Traditional block-based reconstruction simply rebuilds the entire storage from block 0. In contrast, Reo optimizes this process in two aspects.

First, Reo only reconstructs valid data objects, which could be a small portion of the whole storage capacity. Thus, the invalid blocks and irrecoverable objects are simply skipped, which greatly accelerates the recovery process.

Second, Reo prioritizes the recovery based on the semantic importance of data objects. This brings two benefits. First, during the data recovery, another device failure could happen. Prioritized recovery minimizes this vulnerable window by reconstructing the most important data first to create additional data redundancy on the new device as quickly as possible. Second, due to the nature of cache locality, recovering the hot data objects first makes the data objects ready for access at the earliest time, which eliminates the delay of waiting for on-line data reconstruction, improving performance.

Reo offers the highest priority to handle the on-demand access first. When there is no on-demand requests, the reconstruction procedure restores the recoverable data objects according to their class (metadata, dirty data, hot clean data, and finally cold clean data), from Class 0 to Class 3, in that order. In this way, we can realize the differentiated data recovery according to their semantic importance, effectively enhancing data reliability and also performance.

V. Implementation

To evaluate the proposed design, we have prototyped Reo based on *open-osd* [6], an open-source implementation of Object Storage Device (OSD) in Linux. Open-osd has two main parts, *osd-initiator* (client) and *osd-target* (server). The osd-initiator includes an *exofs* module and an *ore* (object raid engine) module, which are adopted in Linux. Our implementation is based on Linux kernel 4.4.10. We have modified *exofs*, focusing on the device table part, and recompiled it into the Linux kernel. For our test, we specify two OSD targets by setting the variable `first_dev` to a fixed value (0 or 1) to specify the operation's target device (OSD0 or OSD1), respectively. For communications between the OSD and the upper-level cache manager, we use a special object (OID: `0x10004`). All writes to this object are performed in synchronous mode.

The osd-target is a user-level program, which is mainly responsible for data object management in the flash array. We have added about 6,000 lines of C code to implement the differentiated redundancy and recovery in an object-based array of flash SSDs in the target. In the original osd-target, all the data objects are managed by the host file system, and the metadata is handled with a SQLite database [11]. In our prototype, the file system and the SQLite database are replaced with our flash SSD array and a hash table to manage the data object storage.

We have also implemented an object-based cache manager for about 2,000 lines of C code on the osd-initiator side. For cache replacement, we use the standard Least Recently Used (LRU) replacement algorithm. The replacement is implemented at the object level. The hot and cold data object classification is also implemented in the cache manager.

VI. Evaluation

In order to evaluate our proposed scheme, we measure the performance of Reo in two situations. (1) *Normal run*. In the normal running state, Reo applies differentiated data redundancy to data objects. It allows us to utilize the cache space more efficiently for achieving a higher cache hit ratio. (2) *Failure resistance*. Upon device failures, Reo enables graceful degradation and provides continuous caching services, though at a degraded performance. Reo also recovers the cache in a prioritized manner by reconstructing the important data first.

In our experiments, we compare Reo with the classic approach, *Uniform Data Protection*, which does not differentiate data objects for parity calculation and recovery and indistinguishingly applies the same level of redundancy to all data objects. We evaluate the effectiveness of Reo in four metrics, *space efficiency*, *hit ratio*, *bandwidth*, and *latency*.

A. Experimental Setup

In our experimental system, we conduct all the tests on three Lenovo TS440 ThinkServers. Each server is equipped with a 4-core Intel Xeon E3-1266 3.3 GHz processor, 16 GB memory, and a 7,200 RPM 1-TB Western Digital hard drive. The cache server is equipped with an array of five 120-GB Intel 540s flash SSDs. The storage server uses a separate 1-TB Western Digital hard disk drive as the back-end data store. All servers are connected in a 10-Gbps Ethernet network. We use 64-bit Ubuntu 14.04 LTS with Linux kernel 4.4.10.

We use MediSyn [36] to generate three representative workloads with various access patterns following Zipfian distributions. We synthesize the workloads by converting the files into objects. Specifically, we generate three group of traces with different localities, namely *weak*, *medium*, and *strong*. All the workloads use a data set of 4,000 unique data objects with the same object distribution. The average object size is around 4.4 MB, and the total amount of data set is about 17.04 GB. The *weak*, *medium*, and *strong* workloads have 25,616, 51,057, and 89,723 read requests in total, respectively. The total amounts of accessed data are approximately 109.4 GB, 220.04 GB, and 386.78 GB for the three workloads, respectively.

We configure the cache size from 4% to 12% of the total amount of workload data set. The three groups of workloads, *weak*, *medium*, *strong*, are used to compare Reo with the traditional uniform data protection, which distributes the parity chunks across the devices in a uniform, round-robin manner and indistinguishingly applies data protection to all the data objects. We use four different settings for the uniform data protection. In particular, *0-parity* means no data redundancy for the entire cache data; *1-parity* and *2-parity* mean that we use 1 parity and 2 parity chunks in a stripe for data protection, and *full-replication* means that we maintain replicas across all devices for all the data objects.

B. Normal Run

An important goal of Reo is to maximize the space efficiency without sacrificing data reliability. Reo selectively applies different levels of redundancy to data objects according to their semantic importance. In contrast, the traditional uniform protection approach simply introduces extra parity information to all data objects equally.

We define *space efficiency* as the percentage of user data among the entire occupied storage space (i.e., the sum of data and parity). Thus, for a five-device flash array, the space efficiency of 0-parity is 100%, and that of 1-parity and 2-parity is 80% and 60%, respectively. Since Reo does not uniformly provide data protection, we reserve a certain percentage of flash space for parity information. Reo-10% means that we reserve 10% of the flash space for parity. For a fair comparison, we also test *Reo-20%* and *Reo-40%* to compare with the two uniform data protection cases, 1-parity and 2-parity. In order to fully exercise the flash cache, the cache server is configured with 4 GB memory in the experiments. The flash array is configured to use a chunk size of 64 KB.

Our results confirm that *Reo-10%* achieves 90.5%, 91.0%, and 90% average space efficiency for *weak*, *medium*, and *strong* workload, respectively. *Reo-20%* and *Reo-40%* also show space efficiency close to the specified parity percentage.

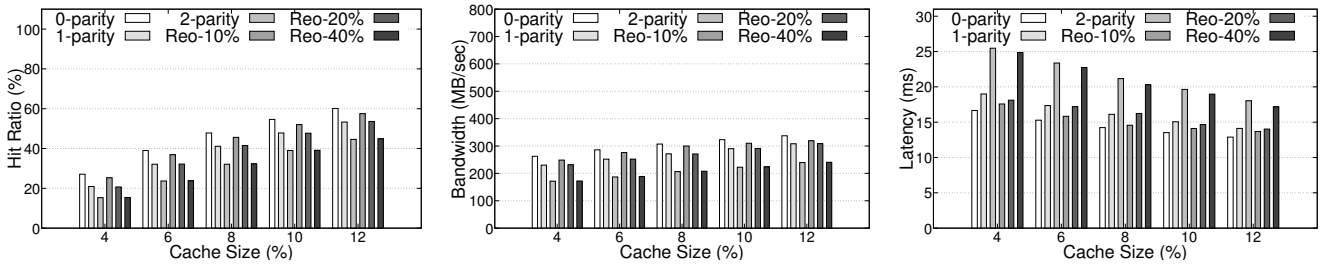
Fig 5 shows the hit ratio, bandwidth, and latency results under normal run. Despite the more complex data object classification and differentiated data redundancy management, *Reo-20%* achieves nearly identical performance to uniform data protection (1-parity) under normal run condition. This is as expected, since they have the same overall space efficiency. Similar results are observed in *medium* and *strong* workload as well. *Reo-40%* shows up to 1.8 percentage points (p.p.) higher in hit ratio than 2-parity, which in turn increases the bandwidth by up to 1.5%, and decreases the latency by up to 2.8%. This is the result of having differentiated data redundancy. Reo does not have to use up the entire 40% flash space reserved for parity, if all the *important* data objects are already protected. Thus, the saved space can be used to cache more data. Also note that this slight performance improvement is as expected, since Reo reserves the same amount of space for parity. This result also well illustrates the low overhead of Reo.

C. Failure Resistance

Reo handles device failures by providing differentiated redundancy, which ensures important data more survivable during failures, to achieve a graceful performance degradation.

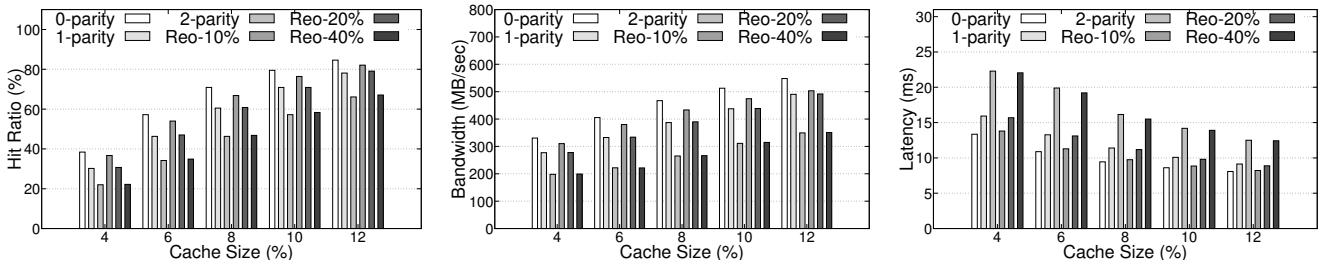
To illustrate the effect of caching services under device failures, we first fully warm up the cache. Then we emulate the device failure as follows. On the *osd-target*, we use an individual program to send the target device ID to the OSD manager via shared memory. Upon receiving the "shootdown" command, OSD takes the device offline and sets all the affected data objects as "corrupted". Applications running on the *osd-initiator* side can check the status through a query command to OSD. If the sense code `0x65` is received, it means that the device failure occurs; if the sense code `0x00` is returned, it means that the object is accessible; if the sense code `0x63` is received, it indicates that the queried data object is corrupted and irrecoverable. In order to ensure repeatable tests, we inject *failure points* in specified positions. We run the *medium* workload and inject four failure points at the 10,000th, 20,000th, and 30,000th, 40,000th requests to trigger four device failures in total, one in addition each time. In the experiments, the cache server is configured to use 4 GB memory and a chunk size of 1 MB. The cache size is set to 10% of the workload data set size.

Upon device failures, the available cache space and the available data in cache both decrease. Figure 8a shows the change of cache efficiency as number of device failures



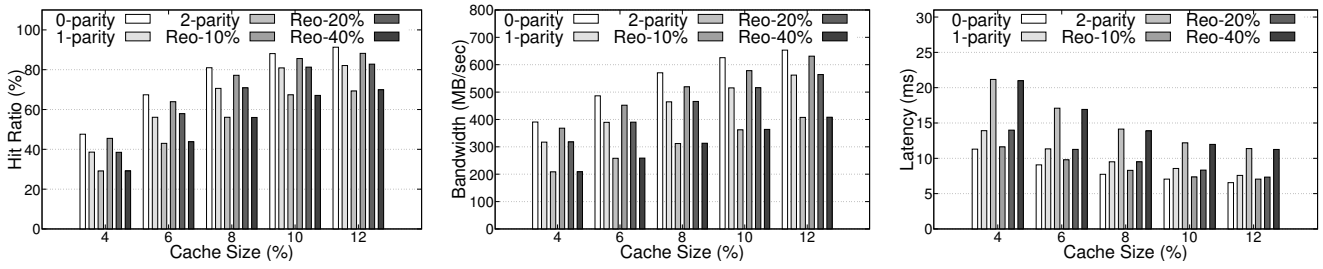
(a) Hit Ratio (b) Bandwidth (c) Latency

Fig. 5: Hit ratio, bandwidth, and latency comparison for weak-locality workloads.



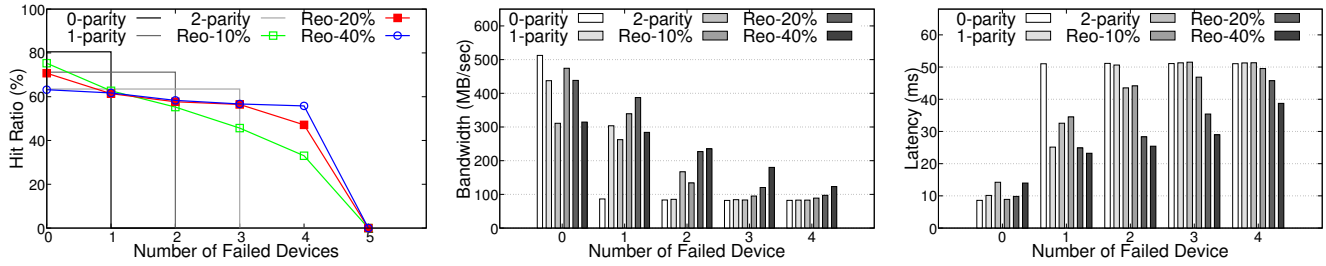
(a) Hit Ratio (b) Bandwidth (c) Latency

Fig. 6: Hit ratio, bandwidth, and latency comparison for medium-locality workloads.



(a) Hit Ratio (b) Bandwidth (c) Latency

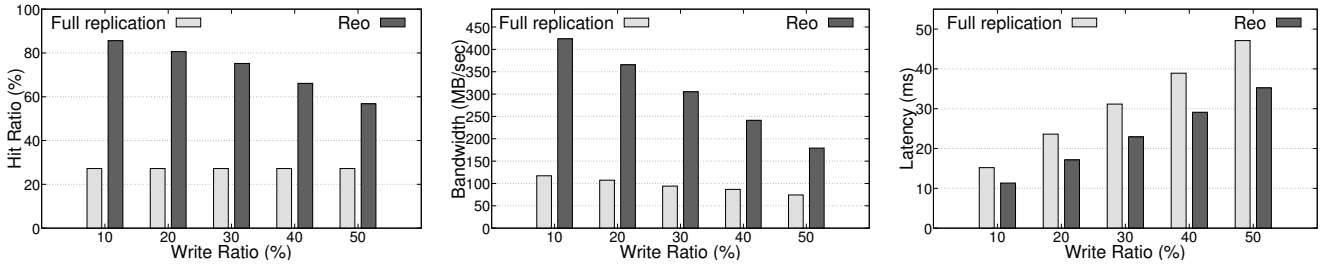
Fig. 7: Hit ratio, bandwidth, and latency comparison for strong-locality workloads.



(a) Hit Ratio (b) Bandwidth (c) Latency

Fig. 8: Hit ratio, bandwidth, and latency during device failures and recovery.

increases. When one device failure occurs, the hit ratio of *0-parity* immediately drops to zero, because there is no data redundancy and the entire cache is corrupted and becomes unusable. In contrast, *1-parity* and *2-parity* are able to reconstruct all the lost data with parity, thus their hit ratios remain the same. The hit ratios of *Reo* are affected due to the loss of the unprotected cold data, but the overall hit ratios remain at reasonable level, since the most important data objects are still accessible. *Reo-10%* shows a hit ratio drop of 12.6 p.p., whereas *Reo-40%* only drops by 1.5 p.p., which is minimal.



(a) Hit Ratio (b) Bandwidth (c) Latency

Fig. 9: Hit ratio, bandwidth, and latency comparison for write-intensive workloads.

We can also find that the larger portion of space is allocated for parity, the less the hit ratio of Reo is affected upon device failures due to the extra data redundancy.

When there is a second device failure, the hit ratio of *1-parity* drops to zero suddenly. It is because with only one parity, the corrupted data on the two failed devices cannot be reconstructed. With the same space efficiency, *Reo-20%* shows a hit ratio of 61.4% when two device failures occurs. We have observed similar results for the case of having three device failures. *2-parity* fails completely, whereas *Reo-40%* still retains a hit ratio of 58.3%. It is worth noting that when more than two devices fail, a cache with uniform data protection, *1-parity* or *2-parity*, becomes completely unusable, with a hit ratio of 0%. While with Reo, cache remains functional as long as there is at least one working device. It is due to the differentiated data redundancy provided by Reo, which gives the highest protection to the most important data. The above experiments clearly show that Reo can achieve graceful degradation, in contrast to a sudden and complete service loss with uniform data protection.

The average bandwidth and latency during recovery are shown in Fig 8b and Fig 8c. Under normal run, *Reo-20%* (438 MB/sec) shows a similar bandwidth as *1-parity* (437 MB/sec). When one device fails, *1-parity* drops to 303 MB/sec, while *Reo-20%* maintains the bandwidth at 387 MB/sec. This is due to Reo's on-demand access and prioritized recovery, with which important data objects still can be accessed directly or reconstructed in the order of their classifications. Similar results are found across all configurations of Reo.

D. Dirty Data Protection

In a write-back cache, dirty data need to be protected to avoid irrecoverable data loss. In this set of experiments, we compare the uniform full replication protection method with Reo. We have synthesized five write-intensive *medium* workloads with the write request ratio varying from 10% to 50%. We configure the cache server with 6 GB memory and use a chunk size of 64 KB. The cache size is set to 10% of the workload data set size.

As shown in Fig 9, without any semantic information, the uniform full replication protection has to assume all the data are dirty, which leads to a low space utilization (20% when using 5 devices) and a cache hit ratio of 27.2%, regardless of the dirty data ratio. Unlike the uniform approach, Reo only saves the replicas of the dirty data objects. As a result of the higher space efficiency, Reo shows a hit ratio of 85.6% when the write ratio is 10%. This in turn makes Reo reach a bandwidth of 423.6 MB/sec, which is 3.6 times of the full replication approach, which only has a bandwidth of 117.1 MB/sec. Even in a worse case with a write ratio of 50%, Reo can still serve 56.8% of the requests in cache, while keeping all the dirty data safe.

VII. Conclusion

In this paper, we present a highly reliable, efficient, object-based flash cache, called *Reo*. Reo is designed to achieve high reliability, efficiency, and performance. By exploiting the semantic knowledge of flash-based object cache, Reo differentiates the data in cache based on their semantic importance, and accordingly provides differentiated data redundancy for maximizing the reliability of important data and differentiated data recovery for accelerating the recovery of caching services. Our prototype based on Object Storage Device (OSD) has shown promising results in experiments.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback and insightful comments. We are also grateful to Boaz Harrosh for his technical advice and detailed support with open-osd. We also thank Ralph O. Weber for providing us the OSD documents. This work was supported in part by the U.S. National Science Foundation under Grants CCF-1453705 and CCF-1629291.

References

[1] Flashcache. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920/

[2] M. A. Roger, Y. Xu, and M. Zhao, "BigCache for Big-data Systems," in *Proceedings of International Conference on Big Data (Big Data'14)*, Washington, DC, USA, October 27-30 2014.

[3] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side Flash Caching for the Data Center," in *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*, San Diego, CA, USA, April 16-20 2012.

[4] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the Robustness of SSDs under Power Fault," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, February 12-15 2013.

[5] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Warming up Storage-Level Caches with Bonfire," in *the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, USA, February 13-15 2013.

[6] open-osd. [Online]. Available: https://github.com/bharrosh/open-osd

[7] Information Technology - SCSI Object-Based Storage Device Commands - 2 (OSD-2). [Online]. Available: https://www.techstreet.com/incits/standards/incits-458-2011?product_id=1801667#jumps

[8] T10 SCSI. [Online]. Available: http://www.t10.org/lists/2op.htm

[9] M. Mesnier, G. Ganger, and E. Riedel, "Object-based Storage," *IEEE Communications Magazine*, vol. 41, pp. 84–90, 2003.

[10] exofs. [Online]. Available: https://elixir.bootlin.com/linux/v4.4.10/source/fs/exofs

[11] SQLite. [Online]. Available: http://www.sqlite.org

[12] H. Zhang, M. Dong, and H. Chen, "Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Santa Clara, CA, USA, February 23-25 2016.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, October 19-22 2003.

[14] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, USA, February 14-17 2012.

[15] H. Weatherspoon and J. Kubiatowicz, "Erasure Coding Vs. Replication: A Quantitative Comparison," in *the International Workshop on Peer-to-Peer Systems*, Cambridge, MA, USA, March 07-08 2002.

[16] J. S. Plank, J. S. Plank, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage," in *Proceedings of the 7th Conference on File and storage technologies (FAST'09)*, Berkeley, CA, USA, February 24-27 2009.

[17] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[18] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write Policies for Host-side Flash Caches," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, USA, February 13-15 2013.

[19] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer, "Flash Caching on the Storage Client," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, USA, June 26-28 2013.

[20] D. Arteaga and M. Zhao, "Client-side Flash Caching for Cloud Systems," in *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*, Haifa, Israel, June 30 – July 02 2014.

[21] D. Qin, A. D. Brown, and A. Goel, "Reliable Writeback for Client-side Flash Caches," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'14)*, Philadelphia, PA, USA, June 19-20 2014.

[22] M. Saxena and M. M. Swift, "Design and Prototype of a Solid-state Cache," *ACM Transactions on Storage*, vol. 10, no. 3, p. 10, 2014.

[23] K. Wang and F. Chen, "Cascade Mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC'18)*, Carlsbad, CA, USA, October 11-13 2018.

[24] F. Chen, D. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*, Tucson, AZ, May 31 - June 4 2011.

[25] J. Liu, "Brief Announcement: Exploring Schemes for Efficient and Reliable Caching in Flash," in *Proceedings of Workshop on Storage, Control, Networking in Dynamic Systems (SCNDS'18)*, New Orleans, LA, USA, October 19 2018.

[26] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, USA, March 31-31 2003.

[27] C. Karakoyunlu, M. T. Runde, and J. A. Chandy, "Using an Object-Based Active Storage Framework to Improve Parallel Storage Systems," in *Proceedings of the 43rd International Conference on Parallel Processing Workshops*, Minneapolis, MN, USA, September 9-12 2014.

[28] B. Hou and F. Chen, "Pacaca: Mining Object Correlations and Parallelism for Enhancing User Experience with Cloud Storage," in *the 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*, Milwaukee, WI, USA, September 25-28 2018.

[29] Y. Kang, J. Yang, and E. L. Miller, "Efficient Storage Management for Object-based Flash Memory," in *Proceedings of 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'10)*, Miami Beach, FL, USA, August 17-19 2010.

[30] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, and S. Maeng, "OSSD: A Case for Object-based Solid State Drives," in *Proceedings of IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*, Long Beach, CA, USA, May 6-10 2013.

[31] Y. Kang, J. Yang, and E. L. Miller, "Object-based SCM: An Efficient Interface for Storage Class Memories," in *Proceedings of IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*, Denver, CO, USA, May 23-27 2011.

[32] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated Storage Services," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, Cascais, Portugal, October 23-26 2011.

[33] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving Storage System Availability with D-GRAID," *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 137–170, 2005.

[34] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD Reliability," *ACM Transactions on Storage*, vol. 6, no. 2, p. 4, 2010.

[35] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the International Conference on Management of Data (ACM SIGMOD 1988)*. Chicago, IL: ACM Press, June 1988.

[36] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat, "MediSyn: A Synthetic Streaming Media Service Workload Generator," in *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video (NOSSDAV'03)*, Monterey, CA, USA, June 01-03 2003.