ObjDedup: High-Throughput Object Storage Layer for Backup Systems with Block-Level Deduplication

Andrzej Jackowski, Łukasz Ślusarczyk, Krzysztof Lichota, Michał Wełnicki, Rafał Wijata, Mateusz Kielar, Tadeusz Kopeć, Cezary Dubnicki, and Konrad Iwanicki, *Member, IEEE*

***Abstract*—The immense popularity of object storage is also affecting the market of backup. Not only have novel backup solutions emerged that utilize cloud-based object storage as backends, but also support for object storage interfaces is increasingly expected from traditional dedicated backup appliances. This latter trend especially concerns systems with data deduplication, as they can offer compelling gains in storage capacity and throughput. However, such systems have been designed for interfaces and workloads that are markedly different from those encountered in object storage. Notably, they expect data to be written in portions that are orders of magnitude longer than those in the novel object-storage-oriented backup applications.**

In this light, we contribute twofold. First, contrasting the properties of object storage interfaces with usage patterns from 686 commercial deployments of backup appliances, we identify specific issues an implementation of such an interface has to address to offer adequate performance in a backup system with block-level deduplication. In particular, we show that a major challenge is efficient metadata management. Second, we present distributed data structures and algorithms to handle object metadata in backup systems with block-level deduplication. Subsequently, we implement them as an object storage layer for our HYDRAstor backup system. In comparison to object storage without in-line deduplication, our solution achieves 1.8–3.93x higher write throughput. Compared to object storage on top of a state-of-the-art file-based backup system, it processes 5.26–11.34x more object put operations per time unit.

***Index Terms*—deduplication, object storage, backup storage, secondary storage**

I. Introduction

OBJECT storage, such as Amazon S3 [1] or Microsoft Azure Blob Storage [2], has become a highly popular and versatile storage abstraction. It organizes unstructured data as objects that are grouped into buckets. Apart from the data themselves, each object normally comprises up to a few kilobytes of metadata, including a key identifying it within its bucket. These storage primitives can be accessed via an HTTP-based interface following REST principles: reading objects/buckets is done with HTTP GETs, uploading with PUTs, deleting with DELETEs, and so on.

The demand for such an abstraction is immense. In 2017, it was estimated that over 30% of data center capacity was in object stores [3]. In 2021, in turn, Amazon alone stored over 100 trillion objects in S3 [4]. Object storage interfaces are provided by hyperscalers [5], [6], other public clouds [7], on-premise enterprise storage systems [8], and open-source solutions [9]–[11]. Likewise, they are utilized in diverse applications, including video services [12], social media [13], and games [14], to name just a few examples.

This success of object storage has also affected the market of backup. There are many properties that make this storage abstraction attractive for these applications. In particular, provided as a cloud service, object storage offers a convenient way of dependably keeping backups off-site, as its interfaces contain provisions for transferring large amounts of data via wide-area networks. They also display design concerns over security, including strong ransomware protection. Consequently, novel backup solutions utilizing cloud-hosted object stores as backends have appeared [15], [16]. Likewise, systems that internally back up their state (e.g., databases or analytic platforms [17], [18]) have added support for object storage. Finally, leading backup applications—originally targeting dedicated backup appliances as storage backends—have started integrating with S3 and similar interfaces [19], [20].

In this light, there is a strong market incentive also to have such dedicated backup appliances implement these interfaces. In particular, appliances that can significantly add much value as backends for object-storage-compatible backup applications are those supporting data deduplication. Deduplication is a technique that can reduce data as many as 10–30 times without any information loss [21]. It is available in a range of storage systems [22]–[24] but one of its main uses is backup. This is because backups inherently contain data that are repeating over time, thereby yielding high deduplication ratios [25]. Notably, an appliance offering global block-level in-line deduplication can reduce storage footprint data written independently by different backup applications, even if backup applications internally implement their own form of deduplication. In combination with simplified regulatory compliance and higher control over data stored by on-premise machines, these compelling space savings and compatibility with cloud-based storage backends are the primary motivation behind adding object storage interfaces to backup appliances.

However, it is unclear how a backup appliance with deduplication should implement such an interface so as to offer ade-

Manuscript received 7 July 2022; revised 18 December 2022; accepted 12 February 2023. Recommended for acceptance by S. Pallickara. *(Corresponding author: Andrzej Jackowski.)*

Andrzej Jackowski, Łukasz Ślusarczyk, Krzysztof Lichota, Michał Wełnicki, Rafał Wijata, Mateusz Kielar, Tadeusz Kopeć, and Cezary Dubnicki are with the LLC, 9LivesData, 02-796 Warsaw, Poland (e-mail: jackowski@9livesdata.com; slusarczyk@9livesdata.com; lichota@9livesdata.com; welnicki@9livesdata.com; wijata@9livesdata.com; kielar@9livesdata.com; kopec@9livesdata.com; dubnicki@9livesdata.com).

Konrad Iwanicki is with the Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, 00-927 Warsaw, Poland (e-mail: iwanicki@mimuw.edu.pl).

Digital Object Identifier 10.1109/TPDS.2023.3250501

quate performance. Normally, for external applications, backup appliances provide dedicated data-transfer interfaces, such as Common Internet File System (CIFS), Virtual Tape Library (VTL), and others [26], which differ significantly from object storage interfaces. Although objects and buckets resemble files and directories from file systems, these primitives do differ. Likewise, object storage is designed for different access patterns than backup appliances. For instance, S3 recommends small objects: if an object is to exceed 100 MB, it should be uploaded in multiple parts for better throughput and recovery from network issues [1]. Accordingly, backup applications for object storage backends typically organize a backup into numerous small objects (1–64 MB) [19], [27], [28]. In contrast, leading backup applications for dedicated appliances use large files ($\geq$100 GBs) [29]. The bottom line is that backup appliances with deduplication may not be prepared to handle the volumes of metadata due to both the specific functionality and usage patterns of object storage interfaces.

In this article, we thus investigate the problem of efficiently implementing an object storage interface in state-of-the-art backup appliances with global block-level in-line deduplication. To this end, we take the following "pragmatic" approach. Since the considered appliances are a mature, complex, and highly optimized technology, we do not aim to redesign any of their internal functionality but only to add new functionality. Likewise, as a single appliance normally offers multiple backup interfaces at once, and data written via any of those is globally duplicated (i.e., also against data written via others), when adding support for an object storage interface, we must preserve this behavior.

Our approach is further reinforced by the fact that many steps of data deduplication are independent of the particular interface, and a backup appliance is itself often a distributed storage system, ranging from a few to as many as thousands of machines. Such a system implements a number of features that work across all interfaces, such as ensuring the quality of service, preventing premature exhaustion of storage space, or controlling data resilience. Implementing these features as a single block-level storage engine that is shared by all exported interfaces is a common practice.

A consequence of this approach is that, similarly to the classic backup interfaces, an object storage interface should be provided as a layer over the block-level engine of a distributed storage system with deduplication. On the one hand, this decreases the number of design decisions required during our study, because effective solutions for many problems are already available. On the other hand, it poses novel problems because adapting some techniques that are popular in object storage without deduplication is not possible, and we needed to propose new dedicated solutions to achieve satisfactory performance.

Given this approach, the major contributions of the article are twofold. First, we present a *preliminary study* that aims to identify particular issues an implementation of an object storage interface for a backup appliance with global block-level in-line deduplication has to address. Based on data from 686 real-world deployments of our backup system, we extract statistical information characterizing their usage patterns. With this information, we analyze commonly used object interfaces to identify requirements, meeting which may be challenging in a system with global block-level in-line deduplication. Second, based on the study, we identify core algorithmic problems and propose our solutions to these problems, that is, *algorithms and data structures*, dubbed ObjDedup, which can be employed to provide object storage functionality efficiently as a layer on top of a block-level engine of a backup appliance. We also outline our implementation of this design for HYDRAstor and evaluate it experimentally. The evaluation indicates, among others, that the presented solutions can outperform the state of the art multiple times in terms of I/O operation throughput.

The rest of the article is organized as follows. Section II provides the necessary background and surveys related work. Section III contains our preliminary study based on real-world deployment data. Section IV introduces the algorithmic core of our solution. Sections V and VI discuss, respectively, the implementation and an experimental evaluation of the solution. Section VII concludes.

II. Background and Related Work

Although deduplication has multiple applications [30], [31], it is particularly appealing for backup and recovery, as consecutive backups often share most of their data. Therefore, it is an important feature in backup systems that utilize mainly HDDs [32], [33]. In this section, we give a high-level outline of the operation of such systems, emphasizing aspects that are the most relevant to our work. A comprehensive overview of general deduplication techniques can be found in an earlier survey [34] and article [35].

A. Global Block-Level In-Line Deduplication

Our work focuses on a model referred to as *global block-level in-line deduplication*, as it is the state of the art for backup appliances. To clarify the reader's intuition, the term "global" captures the fact that data written via any interface and at any time can be deduplicated against each other, thereby yielding higher space savings than local deduplication among data written via a single interface or in a particular backup session. The term "in-line" means that deduplication is performed while the data are to be written to the appliance (i.e., before they are actually written to storage drives), which allows for much higher write throughput of data with numerous duplicates in comparison to off-line/background deduplication (e.g., via post-processing, after the data have been written). Finally, "block-level" denotes that the unit of deduplication is fixed- or variable-length block, typically determined at runtime, rather than entire file, directory, or backup stream, which again improves space savings.

Systems employing global in-line block-level deduplication for backup purposes operate roughly as follows. A backup application assembles the data to be saved by the backup appliance (e.g., files and directories) into a data stream. A deduplication pipeline starts with chunking the stream into blocks, either at fixed offsets or with more advanced algorithms [36]. The resulting blocks typically range from 2 KB to 128 KB [37]. Subsequently, their existence on storage machines running the backup appliance is checked. The decision whether a block is

already present in the system is typically made based on a so-called fingerprint of the block (e.g., its SHA-256 hash). As the aggregated size of all fingerprints can easily exceed terabytes, efficient fingerprint indexing is necessary. Only when the block is confirmed not to be stored, it is written onto the storage drives of selected machines running the appliance.

Deduplication-related research focuses largely on improving the efficiency of backup and recovery. Leveraging Bloom filters [38], flash memory [39], [40], and reorganizing data to reduce fragmentation [41]–[44] are all compeling methods of handling blocks and their metadata. Our research addresses an orthogonal problem: an effective implementation of object storage interfaces on top of a deduplication system that already has its block maintenance optimized. In general, state-of-the-art backup appliances solve numerous problems (e.g., block caching, ensuring the quality of service, balancing space utilization, or controlling data redundancy). Therefore, their block maintenance (including deduplication) is encapsulated into an engine that offers a block-level interface, allowing for reading and writing blocks as well as querying their presence. The higher layers, such as CIFS or VTL, are implemented on top of this block-level engine, treating it mainly as a black box, which is also the approach we follow in ObjDedup.

Consequently, the exact solutions the block-level engine employs to the aforementioned problems are largely abstracted out and should be irrelevant for the higher layers, including ObjDedup. However, the data must be eventually stored by the backup appliance in 2KB–128KB blocks using a well-defined data organization.

B. Deduplicated Data Organization

More specifically, the following data organization is commonly adopted in deduplication appliances and hence is also assumed in the design of ObjDedup.

All blocks are immutable. If block contents were allowed to change after a fingerprint computation, the contents of a block could be lost forever if it was deduplicated against another block that was modified later or, conversely, two blocks with ultimately different contents could have the same fingerprint. The fingerprint of a block is thus normally used as (an element of) the unique address of the block.

Blocks are organized into directed acyclic graphs (DAGs). Since a single block is typically too small to represent an entire data collection, such as a file or a directory, there must be means of grouping multiple related blocks. To this end, besides actual data, a (regular) block can contain addresses of other blocks, thereby referencing these blocks. A fingerprint, and hence the address, of a block is thus computed over both the data and addresses contained in the block. In effect, each data collection can be represented by a (root) block that references the regular blocks holding the data comprising the collection. If the collection is large, this referencing can have multiple levels. The root block also has an address and can thus be referenced from other blocks, for instance, containing backup metadata. In other words, globally, data collections are organized as logical block trees. This organization is only logical because, as a result of deduplication, a block can be referenced by many other blocks (i.e., have multiple parents). Therefore, blocks form DAGs (cf. Fig. 1), rather than trees.

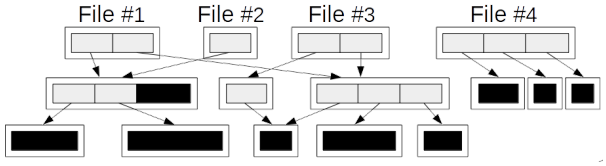

Fig. 1: Data collections (e.g., files) are organized into a set of block DAGs. A block can contain data (black) and/or references to other blocks (gray). Blocks with multiple incoming arrows are deduplicated.

Since block data are immutable and since the address of a block contains a fingerprint computed over both the data and references constituting the block, **the references must be immutable as well**. As a result, changing a reference in some block deep in a DAG entails generating a new block with a new address, replacing references in all its ancestors, so that the change propagates up to the root blocks.

Blocks with no live references are eventually deleted to reclaim storage space. **Deleting blocks in a system with in-line deduplication requires considerable additional effort** to prevent situations where new blocks reference data that have been deleted. Typically, the system employs a multi-phase algorithm that follows a garbage collection technique such as mark-and-sweep or reference counting [45], [46].

C. Deduplication in Object Storage

Despite having a few flavors, object storage interfaces are largely similar [47], [48]: an object storage interface is often described as S3-compatible or Swift-compatible, names originating from Amazon S3 and OpenStack Object Storage (Swift), respectively. In particular, MinIO [11], a popular open-source object store with over 500 million docker pulls, describes its interface as S3-compatible. RadosGW (Ceph Object Gateway) [9], which is in turn an object layer for the widely-adopted Ceph storage, supports both flavors. Finally, even OpenStack Object Storage, the original implementation of the Swift interface, now also incorporates middleware that emulates S3 [49]. Support for either of the interfaces in a backup appliance can thus be extended to other object storage interfaces. However, there has been little work on how such support can be provided efficiently in a backup appliance with deduplication.

To start with, Cloud Tier [50] moves data written to a backup appliance (using an interface different than object storage) to cloud-based object stores. Such object stores do not provide deduplication, so the backup appliance deduplicates data before transferring to the cloud. The solution is much different from ObjDedup, which extends the backup appliance itself with the object storage interface.

Several publications proposed adding deduplication to existing object storage systems, notably Swift and Ceph.

Post-process deduplication approaches, like Ceph's deduplication [23] or LOFS [51], are very different from ObjDedup, which assumes in-line deduplication in the underlying block-level engine. In-line deduplication, if exploited well, can inherently offer superior write throughput and space savings for highly duplicated backup data [34].

In-line deduplication approaches proposed to date do not examine the problem of efficiently managing metadata due to supporting object storage interfaces. In particular, Wang et al. [52] focus on classic files in Ceph and explicitly mention support for Ceph Object Gateway as future work. Similarly, Khan et al. [53], [54] provide deduplication for Ceph's internal objects, which are different from object storage objects, supported in Ceph Object Gateway. In other words, rather than tackling the problems attacked by ObjDedup, that research explains how a variant of the black-box part of ObjDedup (i.e., the block-level engine) could be implemented in Ceph. Those ideas are further improved by CROCUS [55], which schedules deduplication-related operations onto CPUs and GPUs.

In contrast, the aforementioned Ceph Object Gateway requires bucket indexes [56] that are frequently accessed and modified. Therefore, they can incur a significant overhead if kept in a store with in-line deduplication. Yet, we are not aware of any relevant prior performance results for Ceph Object Gateway, and generally, the published results are insufficient to predict how the solutions would behave with large numbers of objects or small files, as generated by backup applications for object storage backends.

Finally, DedupeSwift [57] adds deduplication to Swift. In DedupeSwift, objects are stored as binary files, and metadata are stored in xattrs, so there is no dedicated metadata structure like in ObjDedup. DedupeSwift's throughput tops 10.54–25.51MB/s even with SSDs for deduplication caches, which is insufficient for a commercial backup appliance.

Concluding, we are not aware of any in-depth analysis of the problems posed by a high-performance implementation of an object storage interface for a state-of-the-art backup appliance with global block-level in-line deduplication.

III. Preliminary Study

To gain more insight into the problems, we have conducted a study contrasting real-world usage patterns of backup appliances featuring global block-level in-line deduplication with the relevant properties of object storage interfaces.

A. Object Storage API Analysis

Certain features of object storage interfaces have become a market standard, and understanding what backup applications can expect is a part of our research. Because of space constraints, here we focus only on those features that are the most vital for the considered applications.

When it comes to storage organization, a crucial property is that apart from the data themselves, each object (and bucket) has associated metadata. The metadata of an object contains a key that uniquely identifies the object within its bucket, metainformation on the object's data (e.g., length, MD5 digest), and user-defined metadata. The total size of the metadata can vary and—compared to the size of object data—can be significant. For instance, in Amazon S3, a key is up to 1 KB, and user-defined metadata are up to 2 KB [1].

Likewise, although the basic commands follow REST principles, object storage interfaces are extensive. For example, Amazon S3 currently has almost 100 commands, of which some have no counterparts in POSIX file systems. In particular, besides object management requests, such as PutObject, DeleteObject, ListObjects, there exist commands related to security (e.g., encryption, ACLs, ownership control), multi-part uploads, tiering, replication, and the like. Virtually all these commands access object metadata.

Another feature with far-reaching consequences is the usage of key prefixes. First, objects can be listed given a key prefix and a delimiter. In effect, even though the bucket-object hierarchy has just two levels, a deeper directory-like structure of a classic file system is often recreated by organizing objects through their key prefixes. For example, listing objects with delimiter "`/`" and prefix "`mydir/`" is similar to calling "`ls`" in "`mydir`" of a file system. There are, however, some differences from classic file systems. A major one is that object listings are limited in size (in Amazon S3, to 1000 objects), which entails multiple invocations for prefixes with large numbers of objects. Second, prefixes are utilized for guaranteeing and scaling performance. For instance, Google Cloud Storage initially offers 1000–5000 requests per prefix per second. If the actual number surges dramatically for a prefix, some time may be needed for reorganization, during which the performance is lower [58].

Moreover, as object storage interfaces originally assumed wide-area networks, they promote moving large data in smaller parts. For objects larger than 5MB, multi-part upload (MPU) is recommended, with object transfer split into up to 10,000 parts. MPU state needs to be tracked by the backend because the parts can be provided in any order, and uploads are done in parallel and reattempted if required.

Finally, when operating with object storage interfaces, modern backup applications do make use of their features, notably wide-area-network- or security-oriented provisions. Even though not every application utilizes all commands of an interface, their deep understanding is necessary when developing support for object storage, especially since the market is constantly evolving and the demand is changing. For instance, in recent years, some backup applications started using object locks as a protection mechanism aimed to prevent unintentional data updates and deletes [59], [60]. Another example is that some backup applications avoid MPUs (e.g., by uploading backups in objects below 5 MB [27]), but others do utilize this feature [61]. In either case, however, the typical object size in such applications is between 1 MB and 64 MB [19], [27], [28]. In other words, even if the data fed to a backup application for an object storage interface are the same as the data fed to a backup application for traditional backup interfaces, in the first case, the storage backend will receive data collections that are several orders of magnitude smaller than in the second.

B. Backup Data Pattern Analysis

To contrast these observations on object storage interfaces with the usage patterns of backup appliances with global block-level in-line deduplication, we analyze real-world deployments of such systems. Data is written to a backup appliance in various patterns in backup jobs [40], and the jobs are run periodically (e.g., once a day at a specific time) [62] based on backup life cycles and policies. Even a single backup application can write

hundreds of jobs each week to back up diverse servers and business applications [63]. Similarly, deletion of data from each job is done based on a retention policy (e.g., after five days) [64], but, as mentioned previously, garbage collection under deduplication requires significant work and is thus executed sparingly, at most a few times a week. Therefore, we examine how a system behaves based on information collected over a week, which we refer to as a *sample*. More specifically, we analyze 13,102 samples collected from 686 commercial deployments of our backup appliances and present those results that have had the most impact on the design of ObjDedup. While this is yet to be confirmed empirically when the solutions introduced in this article are massively adopted, we expect that object storage interfaces would not alter the observed patterns significantly, as the backup life cycles, policies, and data themselves are largely independent of a backup interface.

For every sample, we calculate the ratio of the maximal increase and decrease of capacity utilization (cf. Fig. 2). We examine changes in both: *raw* capacity (i.e., data physically stored after deduplication) and *effective* capacity (i.e., data written by backup application, before being deduplicated). The distribution of this value for all samples is plotted in Fig. 3. A majority of samples have their ratios above 1.0, that is, writes exceed deletes, which is expected given the continuous worldwide data growth [65]. Typically, the ratio is in the range 0.66–1.5, so every week similar amounts of data are added and deleted. Samples with no activity are virtually nonexistent, which implies that backups are indeed done regularly.

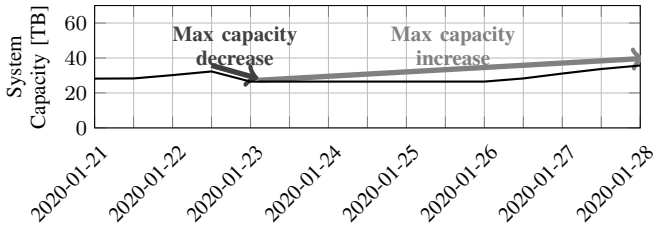

Fig. 2: The evolution of capacity utilization in a representative sample.

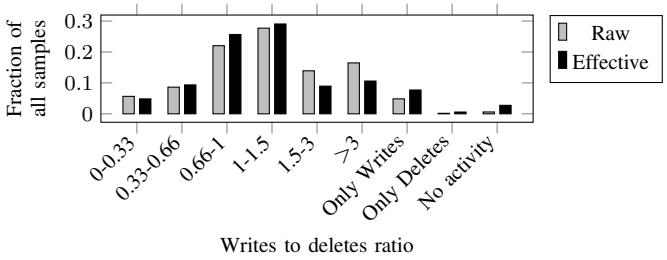

Fig. 3: The ratio of weekly maximal increase and decrease of raw capacity (after deduplication) and effective capacity (before deduplication).

The magnitude of changes to raw and effective capacity utilization is shown in Fig. 4, separately for increases and decreases. It can be observed that capacity utilization in a system can change a lot in a week. The changes in effective capacity have even higher magnitudes than in raw capacity. Given that effective capacity can be larger by an order of magnitude than raw capacity, this means that despite fairly stable capacity utilization (per Fig. 3), the data turnover is considerable: older backups are removed to store fresh ones.

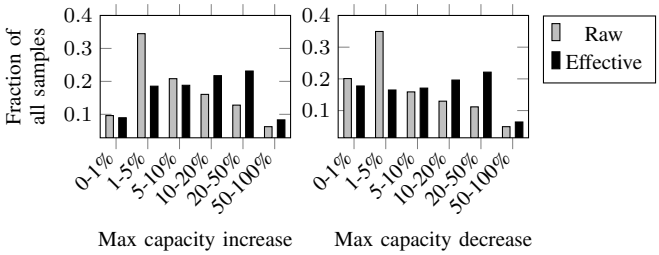

Fig. 4: Maximal positive and negative capacity utilization changes within a week.

Finally, Fig. 5 depicts the distribution of maximal capacity utilization in samples. It shows that although a fraction of free space usually remains, in 16% of cases, the capacity utilization exceeds 80%. Therefore, considering the possibility of substantial ($\geq$20%) utilization increases (Fig. 4), the system indeed relies on efficient garbage collection.

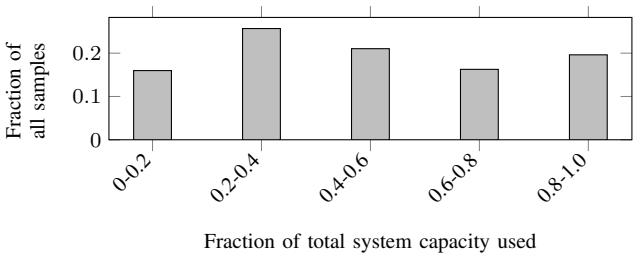

Fig. 5: The weekly maximal utilization of system capacity.

C. Main Lessons Learned

The following major lessons can be drawn from our study.

Backup appliances and object stores have dissimilar characteristics. Backup appliances are optimized for write throughput, which is crucial given that their overwhelmingly dominant usage pattern is writing long data streams. In turn, operating on individual data collections hardly ever takes place as even recovery typically concerns entire snapshots and is considered rather sporadic; the same applies to modifying metadata of existing collections. In contrast, object storage interfaces have been designed for flexibly organizing, efficiently accessing, and remotely managing large numbers of data items, so as to cover many use cases that may be encountered in the plethora of possible cloud-oriented applications. To this end, object storage features an extensive API, rich system- and user-defined metadata enabling this API, and additional provisions for wide-area communication, scalability, security, and the like. It also requires respecting peculiar constraints regarding shaping the organization of the data and access traffic.

Dynamic and relatively large metadata of an object are problematic under immutable deduplicated blocks. For an object, the size of its metadata can be significant compared to the relatively small size of the data themselves, as recommended by object interfaces and respected by backup applications for object stores. In backup appliances, this phenomenon is likely to be aggravated since the data are normally repeating over

time, and hence are often deduplicated. Furthermore, their high weekly replacement rate implies that many objects will be written and deleted every week. Each such operation on an object also requires at least one access to its metadata. In general, many object operations solely affect the metadata, updating them in some way. This is problematic given the block immutability in backup appliances with global block-level in-line deduplication. What is more, some object storage operations require tracking their progress by the backend. Such an operation generates metadata that are heavily accessed for a short time and are deleted afterward but need to be kept persistently in the backend to allow completing the operation even under transient failures. A prominent example is the aforementioned multi-part upload (MPU), which can produce thousands of metadata items for a single object.

Efficiency of metadata management is a fundamental problem on the scale of an entire backup appliance. This is due to the assumptions object storage interfaces make regarding the use of object key prefixes for collective operations, such as object listing, and for performance scaling. In particular, the listing feature implies that the metadata of all objects should be somehow indexed or sorted for efficiency. The potential solutions are further constrained by the fact that the delimiter of subsequent prefix parts is provided on demand and can thus be arbitrarily changed at runtime, even between requests. Guaranteeing performance and scalability, in turn, requires dynamically distributing the load on various objects between machines, for instance, based on the indexes. This implies that the algorithms for managing object metadata have to be able to work in a distributed fashion and handle partial failures.

Object metadata management solutions must not impair the performance of space reclamation. As revealed by our study, the high weekly data replacement rate in backup appliances already entails extensive use of block deletion and garbage collection. Supporting an object storage interface will likely increase the pressure on these mechanisms. This is because any update to metadata stored in immutable blocks typically invalidates these blocks as blocks with the new version of the metadata are written. Therefore, when addressing the previous problems, any implications on space reclamation must be carefully considered so that its efficiency is not impaired. In particular, adding an object storage interface to a backup appliance must not lead to situations in which blocks that are no longer necessary are not garbage-collected as soon as possible because of some dangling references, for instance, due to metadata indexing.

IV. The Design of ObjDedup

In this section, we translate the conclusions from our preliminary study into algorithmic problems and present solutions to these problems, which we dubbed collectively ObjDedup.

A. Problem Statement

As explained previously, a backup appliance with global block-level in-line deduplication typically exports multiple well-established interfaces, that are utilized by external backup applications, possibly at the same time. Internally, in turn, it is usually implemented as a distributed system that encapsulates the core functionality of deduplicated write-optimized fault-tolerant storage into a block-level engine, which is often a product of many years of development and fine tuning. The external interfaces are simply implemented as higher layers on top of this shared engine. We thus assume the object storage interface to be provided in the same manner. This assumption imposes a few constraints on our solutions, the major ones being:

1) The block-level engine must not be changed so as to avoid affecting the operation of the other interfaces exported by the appliance.
2) Likewise, extra hardware, such as additional machines or custom storage devices, must not be required from the appliance to support the new functionality.
3) The performance of the object storage interface, notably write throughput, space utilization, and fault tolerance, must be comparable to that of the classic interfaces.

Under these constraints, we consider the following overall design of ObjDedup. Objects and buckets are organized as other data collections (e.g., files and directories): into logical block trees within the block storage, with the root block representing a particular object or bucket and regular blocks holding the data of the object/bucket (cf. Fig. 1). In effect, the existing, highly-optimized pipeline can be utilized for writing object and bucket data, which allows for ensuring the same performance of these operations as for the other interfaces of the backup appliance. Object/bucket metadata are also kept in regular blocks within the block storage. Although an alternative design involving dedicated hardware for the metadata, like SSDs or NVMs, could improve the performance of operations on the metadata, it would violate the previously formulated constraints. Moreover, storing the metadata within the block storage is essential for fault tolerance: if a machine responsible for a particular portion of the metadata fails, other machines can take over, as the block engine ensures that metadata are stored redundantly in the block storage and are available to all machines comprising the appliance. In contrast, what is different in the case of metadata compared to data is that because of the way object storage uses key prefixes in multiple operations and performance scaling, the metadata must be indexed and/or sorted by object/bucket key prefixes.

The central algorithmic problem that has to be solved can thus be formulated as follows:

> *How to efficiently organize object and bucket metadata by key prefixes and dynamically manage this organization by multiple processes given shared deduplicated write-optimized fault-tolerant immutable-block storage?*

Efficiency in this context has two facets.

First, the achievable throughput of the object storage interface must be comparable to that of the classic interfaces of the backup appliance, ensuring among others that while accesses to metadata are write-optimized, the performance of reads is not impaired. More specifically, as HDDs are assumed as the main storage medium, achieving a high write throughput is possible only if random disk I/Os are limited. Since blocks are immutable, updating metadata structures requires both reading

some already stored blocks and writing new ones. Whereas block-level engine batches writes, random-read I/Os may easily exhaust HDD capabilities. Therefore, our asymptotic complexity goal for the number of reads necessary to update the metadata of an object/bucket is $O\big(log(n)\big)$, where n is the total number of objects and buckets in the store. We also want to ensure that in the case of updating the metadata of u objects sharing the same prefix, the complexity is $O\big(log_s(n) \cdot \frac{u}{s}\big)$, where s is the expected number of object metadata entries per block.

Second, the storage space of the appliance must be used efficiently as well. Not only does this mean that the storage overhead on metadata should be limited, preferably to $O\big(log(n)\big)$, but also, what is particularly important in a system with deduplication, that deleted blocks containing references to other blocks should be garbage-collectible without an unacceptably long delay. To be more specific, at any time, the number of blocks that store lifeless references should be smaller than a constant M and the constant itself should be small enough to ensure that in practice blocks can be updated within seconds or, at most, minutes.

Last but not least, our formulation of the problem entails that the management by multiple processes of the logical structure holding metadata in the block storage must be resilient to failures of these processes, so that the resulting solution can be made as fault-tolerant as the underlying block storage itself. This necessitates distributed algorithms.

B. Principal Ideas

To address the problem, we analyzed or experimented with multiple potential solutions: from database-oriented or file-system-oriented data structures and algorithms for write-once or erase-before-write storage drives to various fault-tolerant distributed indexes [66]–[70]. In short, the fact that the blocks in the assumed underlying storage are immutable and organized into DAGs limits the applicability of techniques that employ in-place updates, notably classic B-tree or many of its modern variations [66]. In our settings, these techniques would be inefficient because emulating each in-place update would require rewriting not only the updated block but also its every ancestor in the DAG. A particularly promising data structure for immutable-block storage was LSM-tree [67], which is widely adopted in distributed databases and offers an excellent amortized cost of insertions. However, to this end, it requires keeping deleted elements for indefinite periods, which is at odds with the need for promptly garbage-collecting deleted data. According to our preliminary study of backup deduplicating systems, keeping just a single deleted block with references can prevent reclaiming multi-gigabytes worth of storage space. This can be very problematic, even if it happens just for few days.

All in all, we were unable to find an existing solution that would fit the assumed model of block storage with deduplication while at the same time being able to maintain the massive amounts of metadata required by object storage. Consequently, we have devised new data structures and algorithms dedicated for the considered scenarios.

More specifically, our solution involves two persistent data structures dubbed ObjectMetadataLog (OML) and ObjectMetadataTree (OMT). From the systems perspective, they are used to store object metadata and live references to object data in a write-optimized fashion: all metadata updates are first appended to an OML and only asynchronously (in the background) applied in batches to an OMT, which decreases write latency and improves throughput while at the same time ensuring efficient indexing by key prefixes. Both structures are kept in the deduplicated write-optimized fault-tolerant shared immutable-block storage, and their parts are also cached in memory. There is one instance of OMT in the storage and as many instances of OML as there are processes implementing the object storage interface. For scalability, this number of processes can be dynamically controlled to ensure, among others, appropriate collective throughput and failure resilience. For presentation purposes, however, let us assume for a while that there is only one such process. We will drop this assumption shortly.

C. Object Metadata Log (OML)

The OML contains a sequence of *yet unapplied* metadata updates ordered by their time of arrival (cf. Fig. 6). In particular, it also keeps object removals because, for instance, combinations PUT-then-DELETE and DELETE-then-PUT differ in their outcome. A new operation modifying metadata is appended to the sequence, and the client is notified as soon as the append completes. To further improve the throughput of metadata writes, a burst of operations can be batched into a single write request to the block storage. Moreover, the sequence is kept small enough so as to be stored not only in the block storage but also in a memory buffer.

When the in-memory buffer is filled, it is swapped with a new buffer, and the operations it contains are applied to the OMT in the background. The application procedure has to be sufficiently fast so that the memory available for the buffers is not exceeded in the meantime. If that happened, processing client commands via the object storage interface would have to be paused, heavily impairing the overall write throughput. We will address this issue shortly.

When the operations from an in-memory buffer are applied to the OMT, the buffer is ready to be reused, and the OML blocks corresponding to its contents are also deleted from the block storage (i.e., marked for garbage collection). Apart from potential memory constraints, this is another reason for keeping OML in-memory buffers small: in the block storage, the contents of such a buffer may include references to blocks that may be suitable for garbage collection (e.g., root blocks of deleted objects) and hence after an application to the OMT, they should be deleted as quickly as possible to reclaim storage space.

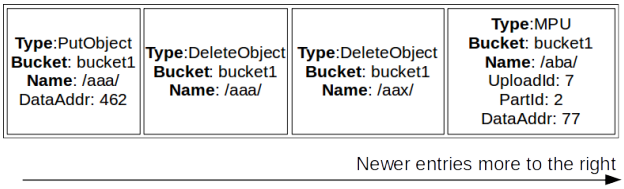

Fig. 6: Sample contents of an OML.

The reader may have noticed that the OML is never read from the block storage during regular operation, as the in-memory buffers are sufficient. However, keeping the OML also in the block storage is necessary for fault tolerance. If a process running the object storage interface fails (e.g., its host machine crashes), it (or another process) can continue after recovery without losing any updates to the metadata.

D. Object Metadata Tree (OMT)

The OMT complements the OML by organizing the object metadata in the block storage to enable efficient access, notably looking up and listing by key prefixes. Unlike the OML, the OMT is meant to be very large, as it keeps most metadata of the system—possibly for hundreds of millions of objects. The OMT resembles a B^+*-tree* and keeps a few types of metadata utilized in object storage in its leaves. The most important ones are metadata of individual objects, MPU parts, and buckets (cf. Fig. 7). They are sorted by their type, key, and other content. Apart from storing the metadata, OMT leaves also have references to the roots of the logical block trees with actual object data.

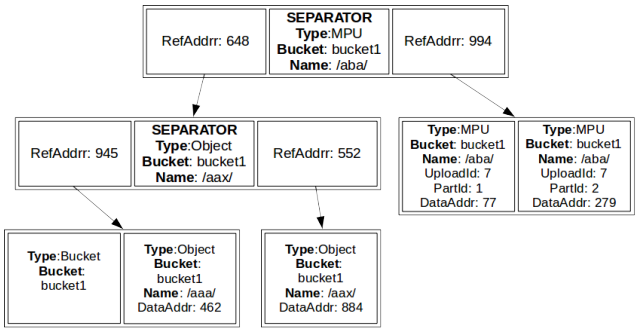

Fig. 7: An example of an OMT.

The blocks that store internal nodes of the tree have references to the blocks with next-level nodes and separators. Each tree node, except for the rightmost ones, has its size between $\frac{1}{2}S$ and S elements, where S is a configuration parameter. To keep the tree balanced, all paths from the root node to the leaves, apart from the rightmost one, have the same length. OMT merges are the only operation that modifies the OMT and they keep both aforementioned invariants. All OMT operations are listed in Table I.

In contrast to the OML, the OMT is too large to fit in memory. However, a subset of its nodes is cached to improve performance. In particular, as our preliminary study shows, the list operations limit the number of returned objects, so caching internal nodes can significantly accelerate consecutive listings. The size of such a cache is meant to be small: asymptotically proportional to the maximal size in the in-memory buffer of the OML.

TABLE I: List of OMT operations.

	Description
OMT Merge	Applies changes from an OML to the OMT (described in Section IV-E)
OMT Distributed Merge	Applies changes from an OML to the OMT in a distributed manner and consists of two phases: SubOMT Generation (Section IV-G1) and OMT Combining (Section IV-G2).
OMT Lookup	Searches for an object in the OMT from its root node to a leaf (B-tree search).
OMT Prefetch	A special lookup version optimized for OMT merges (Section IV-F).

E. Metadata Merge

Metadata merge is a background operation of applying to the OMT all changes from the OML. Since blocks holding the tree nodes are immutable, conceptually, the operation has to generate a new tree. However, rewriting all OMT nodes each time an in-memory buffer of the OML fills up would be an overkill. In particular, it would entail rewriting the metadata of every object and bucket in the system. Therefore, instead, the nodes from the old tree are reused whenever possible. In contrast, the blocks containing the overwritten nodes are eventually not reachable from any live blocks, and can thus be garbage-collected.

More specifically, a merge traverses the OMT in a depth-first search (DFS) manner. For each node, a decision is made if either the whole subtree of the node can be reused in the new version of the tree or some update exists in the OML that has to be applied in the subtree. If such an update exists, the subtree is traversed recursively down to locate the relevant leaf node. If, after the update, leaf size is not in $[\frac{1}{2}S, S]$, the node is split into two or the node next to the right is read so as to combine the two nodes into one or more nodes of valid sizes. In any case, all internal nodes on the path from the rewritten leaf to the root require rewriting as well, because the block addresses of the nodes deeper in the tree have changed. For a rewritten internal node with invalid size, splitting or combination is done as for the leaf. If the size of the root node exceeds S, the node is split, and a new root node above is added: the tree grows by one level.

Special care is given to the MPU delete operation, whose single entry in the OML affects up to 10,000 parts of the deleted MPU in the OMT, and hence could potentially be costly. In such a case, only the two leaves containing the start and the end of the MPU range (and their OMT ancestors) need rewriting, while the nodes in between become suitable for space reclamation. This is because, after the OMT rewrite, their blocks are no longer referenced, directly or indirectly, from any live blocks, and hence will be garbage-collected eventually. In this way, instead of up to 10,000 nodes, the MPU delete affects only a number of nodes proportional to the height of the OMT.

F. Metadata Merge Prefetch

As mentioned previously, the pace at which the metadata merge operation can be done is crucial for the entire system's performance. A metadata merge that iterated through the OMT and issued a new read to the block storage each time a node intersected with an entry from the OML would last far too long and hence could lead to the aforementioned OML in-memory buffer exhaustion. This, in turn, would require pausing user operations, thereby severely deteriorating the overall backup throughput.

As a remedy, we thus propose a prefetch algorithm, referred to as *OMT Prefetch*, that reads from the block storage at most $2h(b-m)+3h\cdot m = h(2b+m)$ OMT nodes, where h is the OMT height, b is the length of the OML in-memory buffer (in entries), and m is the number of MPU delete entries in the buffer. The algorithm can be run in parallel for all entries of the OML in-memory buffer, so at most h sequential steps (i.e., causally-dependent reads) are required to prefetch all nodes. In other words, the work and span of the algorithm are respectively $h(2b+m)$ and h.

The prefetch distinguishes three types of OML operations: metadata inserts/updates, object deletes, and MPU deletes. For each type, a different set of nodes is prefetched. First, a metadata insert or update can overfill a leaf node and may thus force splitting it and possibly its ancestors. Splitting a node does not require reading any other nodes, so for an insert/update operation in the OML, only those nodes are prefetched whose key ranges include the inserted/updated key. A metadata delete can in turn lead to combining or combining-and-splitting a leaf node, and possibly its ancestors, with the first succeeding nodes at the same levels. Therefore, only the nodes whose key ranges include the deleted key and their first right siblings are prefetched.[1] Finally, as explained previously, an MPU delete can lead to removing multiple nodes and updating no more than three nodes at each level. It is thus enough to prefetch the nodes that intersect with the keys representing the start and the end of the MPU parts, together with their ancestors, and the first right siblings of the nodes on the MPU end path (all of the nodes in between will be deleted).

G. Distributing Metadata Merge

The solution described hitherto assumes only a single process operating on the OML and OMT. However, the number of such processes must be scalable to handle more load and tolerate failures. A straightforward approach would be to partition the buckets among the multiple processes so each process would operate on a disjoint set of buckets and objects they contain. However, this may lead to overloading the processes responsible for popular buckets.[2]

We propose a solution in which, rather than only buckets, also individual objects are partitioned among the processes. Metadata operations for a given object are directed to the corresponding process. Each process appends updates to its objects and buckets into its private OML. However, the OMT is shared by all processes, which requires distributing the previously described metadata merge operation. Such a distributed merge proceeds in two phases (see Fig. 8). First, many disjoint OMTs, called SubOMTs, are generated by individual processes. Second, all these subtrees are combined into a single OMT, using a parallel algorithm.

[1]With a small exception: the sibling of the leaf, which is not prefetched even though it may be needed to create a new leaf of proper size. The reason is that such a read can be done on demand later without affecting the critical path and the tree iteration, and we wanted the algorithm never to read more leaves than necessary.

[2]As a side note, in theory, rather than a bucket, a particular object could be popular and receive excessive load. Our solution does not aim to address this simply because, in practice, we have not observed this phenomenon to be relevant to the backup use case.

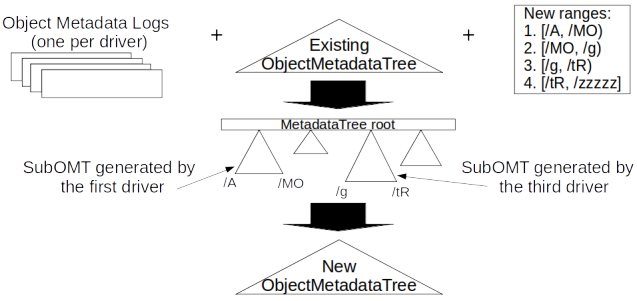

Fig. 8: The phases of the distributed metadata merge.

1) SubOMT Generation Phase: To distribute work evenly, the space of OMT keys is divided into ranges, and each range is assigned to a different process. To achieve this, keys are first partitioned among all processes (e.g., based on their strong hashes), so that each OML is expected to have a similar size. Before each merge, a single, dynamically chosen process calculates boundaries for the ranges based on the contents of its OML. Then, the ranges are broadcast to the other processes, so that each process can generate its own SubOMT by merging relevant entries from *all* OMLs with the subset of OMT nodes that are within its assigned range. In this phase, each process reads all OMLs, but this happens simultaneously and, per previous explanations, the OMLs are small, and hence the block storage caches can be effectively used.

2) OMT Combining Phase: After the first phase, each of the resulting subtrees covers a disjoint contiguous key range. In the second phase, the subtrees are combined into the new global OMT with all keys. This process is not trivial because the nodes on the rightmost path of each subtree can have their sizes below $\frac{1}{2}S$ and the heights of the subtrees may vary.

For two subtrees, we present how to generate a valid OMT by reading only the rightmost path of the first subtree and the leftmost path of the second (also see Fig. 9). First, read the rightmost path of the first subtree. According to the invariant, all other nodes in the subtree have correct sizes. Likewise, read the leftmost path of the second subtree. Starting from the leaf node in the leftmost path of the second subtree, add each entry from the node to the corresponding node from the rightmost path of the first subtree. If the size of the node is exceeded, create a new node and add a reference to it in a higher-level node; if the size of that node is exceeded too, repeat the process. The key observation is that if there are two nodes and at least one of them (the one from the second tree) has a valid size then one or two valid nodes can be created in a way that there are no leftovers. Ultimately, the only nodes with their sizes less than $\frac{1}{2}S$ are: the node that contains the entries from the root of the second subtree, the nodes on the rightmost path of the second subtree (it was spliced), and, if the first subtree was higher, the upper nodes on the rightmost path of the first subtree. Altogether, the combining reads only h_1+h_2 nodes, where h_1 and h_2 are the subtree heights. The tree height is tiny, as discussed shortly, and the appropriate paths in different SubOMTs can be read in parallel from the block storage, so even for huge numbers of keys, one process is sufficient to combine thousands of SubOMTs quickly.

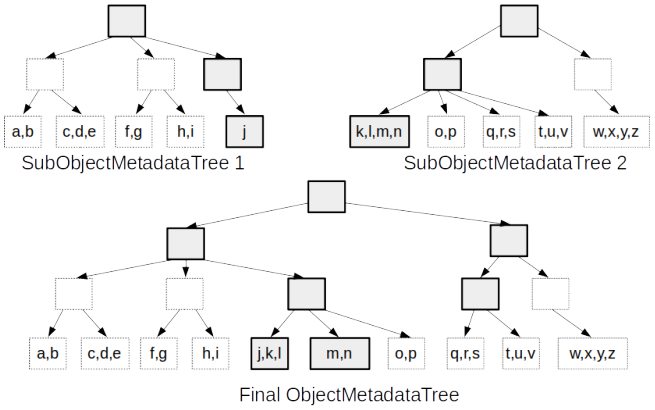

Fig. 9: Combining two SubObjectMetadataTrees with S = 4. Only the gray nodes are read and written. The letters denote object keys.

3) Remarks on Object Key Space Partitions: The presented solution spreads the load due to handling metadata between all processes, for instance, based on hashes of keys. Such an approach is efficient for those object storage interface commands that affect a single key (e.g., GET, PUT, DELETE), because one process can handle a given command invocation entirely. Interestingly, an MPU command invocation can also be handled by one process because the number of parts in such a request is limited (e.g., to 10,000). However, an object interface does contain commands that read information about many keys at once, like object listing. Efficient and consistent handling of such collective operations requires special attention.

Since the OMT is global and maintains the order of keys, retrieving metadata necessary, for example, for object listing from the OMT requires just a handful of reads, and hence they can be performed by one or multiple processes. Object storage interfaces limit the size of an object listing (e.g., to 1000 objects) and, in general, of the output of similar collective operations. In effect, many consecutive requests must be sent to generate a longer output. Nevertheless, such long outputs are also handled efficiently, as the block storage can cache the blocks corresponding to the repeating node paths in the OMT. The issue, however, is that the freshest metadata are not stored in the OMT but in the OMLs. Therefore, during a listing or a similar collective operation, the metadata from the OMT must be updated with the metadata from the OMLs of relevant processes. If the metadata are distributed by key hashes, virtually every OML must be contacted, which entails flooding all processes with requests.

To avoid flooding, we propose to dynamically partition keys among processes. More specifically, the space of keys can be divided into ranges that are dynamically calculated based on the current load. If there are few or no requests, each process can be responsible for a similar number of keys. It reports its load to a distinguished process, so that if one process receives more requests than it can handle, the ranges can be recalculated. If necessary, multiple processes can handle the same range, and in such a case, requests within that range can be further distributed to selected processes based on the key hashes. With this approach, collective operations read metadata handled by multiple processes only when necessary, and only the relevant subset of the OMLs is affected.

4) Failover Handling: In the case of a process failure, another existing or new process can take over. This requires only restoring in-memory data that are kept persistently in the block storage: the in-memory buffer of the OML of the failed process and the cache containing the OMT nodes with the operations from the buffer applied. If the failover is handled by some existing process instead of a new or recovered one, the process also needs to take over the metadata merge responsibilities for the failed process, so that an ongoing merge can be completed. In effect, it has additional keys to handle during the merge. This temporal imbalance is naturally corrected by a new key partitioning, which will take into account the change and distribute the work more evenly.

H. Final Remarks

The proposed solution can be used even in very large systems. The height of the OMT is limited by $h = \lceil log_s(n) \rceil$, where s is a branching degree (limited by constant S) and n is the total number of objects and buckets. Even for a massive 20,000-machine backup appliance, with each machine having 12 high-end 14-TB HDDs, an average object size of 10 MB, 20:1 deduplication ratio, and 5:1 compression, there can be $n =\sim 3.36 * 10^{13}$ objects. With $S = 64$, which is reasonable to keep the OMT nodes small when keys are large (i.e., 1 KB), the tree has at most 9 levels.

Further calculations confirm that keeping metadata in block storage rather than, for instance, in memory or on dedicated local SSDs of machines hosting the processes implementing the object storage interface is not only a design decision but actually a necessity in large systems. For example, in the previous system, each machine needs to store metadata of $1.68 * 10^9$ objects, so with 1 KB keys, they take 1.7 TB. If the objects are smaller (e.g., 1 MB) and have additional 2-KB user-defined metadata, the required capacity per machine sums up to over 50 TB. Assuming a typical hardware architecture of backup appliances, storing such a volume of metadata in RAM or on SSDs is infeasible today. Even if there are flash drives in such appliances (which is not always the case), their capacity can already be used for other purposes. In other words, our algorithmic assumption is reinforced by the limits of today's technology.

V. Implementation

We have implemented ObjDedup in the aforementioned HYDRAstor system [22]. At the time of writing this article, it was part of the product, delivering the object storage interface in the same way as the classic backup interfaces.

A. Overall Architecture

From a systems perspective, HYDRAstor consists of storage nodes, which keep data on their drives, and a layer of access nodes, which provide external access (cf. Fig. 10). The number of storage and access nodes can vary depending on the capacity and performance targets, and the system can scale from one server to multi-rack installations.

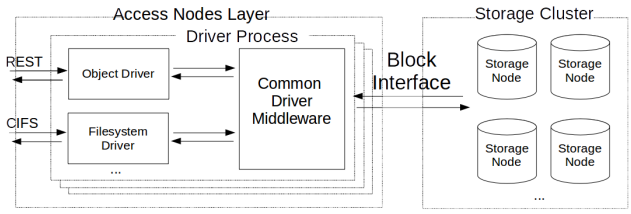

Fig. 10: High-level architecture of access and storage nodes. One or multiple instances of driver processes communicate through a block interface with the storage cluster.

Storage nodes maintain data by means of the block-level engine. In particular, they manage the distribution of blocks among storage media and memory caches, perform operations on the blocks, handle hardware failures, coordinate garbage collection, and the like. Storage nodes compose a storage cluster that exports a coherent block-level interface to access nodes, with operations like writing a block or querying if a block is stored given (a hash of) its content.

Access nodes, in turn, provide higher-level interfaces, like CIFS or, in our case, REST, on top of this block-level engine. These interfaces are implemented as drivers. Internally, they use common middleware that facilitates reusing functionality for accessing the block-level engine of the storage cluster. Especially the deduplication pipeline, including chunking and fingerprinting, is implemented in that middleware and coordinated by the access nodes. Likewise, services for distributing computations, including optimized message routing (conceptually similar to MPI [71]) and locating nodes, are provided by that middleware.

Overall, this architecture matches what we assumed previously for our algorithms. In this view, our work concerns the *object driver*, which implements the algorithms for OML and OMT to provide an object storage interface on top of the common driver middleware (cf. Fig. 10). Depending on the scale of the system as well as client-specific performance and fault tolerance requirements, instances of the object driver are hosted by one or more access nodes.

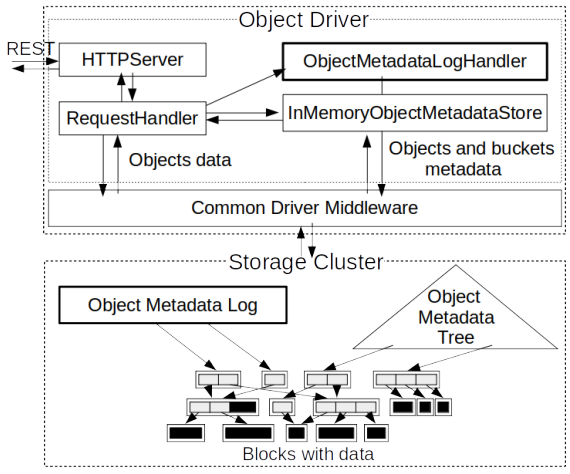

Fig. 11: The architecture of the object driver.

B. Object Driver Architecture

The most outer layer of the object driver is HTTPServer, which receives REST commands (see Fig. 11). They are then processed by RequestHandler, which implements the logic of handling both data and metadata of objects and buckets, which ultimately end up in the storage cluster. For data-related operations, RequestHandler essentially uses the aforementioned common driver middleware, as data can be handled similarly to the other interfaces. For metadata-related operations, it also collaborates with ObjectMetadataLogHandler and InMemoryObjectMetadataStore. ObjectMetadataLogHandler maintains the in-memory buffer of the OML corresponding to the instance and is responsible for coordinating metadata merge operations. InMemoryObjectMetadataStore manages the cache of the OMT nodes with the updates from the in-memory buffer of the OML applied.

Again, this architecture is coherent with our algorithmic assumptions. HYDRAstor is write-optimized, and thus the object driver must follow the same principle. Writing data is inherently optimized by the storage cluster, and the control flow in the driver does not add any extra steps. For handling metadata, in turn, the object driver simply employs ObjDedup, which is also write-optimized by design.

C. Additional Issues

To reduce the latency of selected read and write operations, especially on small files or during more interactive sessions with the backup appliance, HYDRAstor features priority requests. They have a higher preference in queues, and their outcomes are reflected in storage media faster. However, they must not be abused because their performance gains would not be observable in such a case. Following this approach, our object driver uses priority requests only for appends to the OML to further improve the client-perceived latency of metadata write operations. In contrast, all other requests to the storage cluster are regular (non-priority) ones. This is possible thanks to the fact that metadata merge is done in the background and utilizes OMT Prefetch, which anticipates and parallelizes future reads, thereby making metadata merge fast even without priority requests. In the same way, OMT Prefetch improves the performance of collective operations, like object listing by the key prefix.

Such efficiency is also important for space reclamation. As a basis for garbage-collecting dead blocks, HYDRAstor utilizes reference counting. The space reclamation process is done in the background, in parallel to normal requests. Because of deduplication, parallel requests can increase or decrease block reference counts while seemingly dead blocks are being garbage-collected. The space reclamation algorithm must remain correct in the face of such concurrency, which is not trivial [45]. In particular, the algorithm operates in epochs and imposes restrictions on driver-kept block addresses. Notably, in epoch T, a driver must not keep addresses obtained before epoch $T-1$. From the perspective of the object driver, if the deletion operation for an entity (e.g., an object, bucket, MPU) is not applied from the OML to the OMT and deleted from the OML, a reference to the entity is live and garbage collection does not remove the entity. In practice, this means that the object driver

should be able to apply changes from the OML to the OMT within a few minutes. This reinforces our previous claims about the need for the efficiency of metadata merge in ObjDedup.

Another side issue is the use of a dynamically selected, distinguished object driver instance in cases when multiple such instances operate in parallel. Essentially, this problem entails leader election. As HYDRAstor already implements leader election and automatic failover, we were able to reuse this functionality. In practice, however, any sensible at-most-one leader election algorithm could be used instead [72].

Finally, object storage interfaces are sizeable and change over time. One cannot simply ignore some of their functionality because its use by backup applications evolves as well. Therefore, while devising the algorithmic solutions was already challenging, implementing them as the production-ready object driver without altering the block-level engine was equally demanding. In effect, however, as the requirements of ObjDedup on this engine are minimal, it can likely be implemented in other systems with interfaces allowing for writing immutable blocks in tree-like structures, which is a common feature in deduplication storage [34].

VI. Evaluation

To evaluate our solutions, we have conducted numerous experiments using the implementation of ObjDedup for HYDRAstor. We present the most important results in three groups: Section VI-A, containing experiments that evaluate the main performance goals of ObjDedup in a distributed setup, Section VI-B, offering a comparison with the state of the art, and Section VI-C, encompassing a detailed evaluation of OMT performance in various scenarios, which aim to highlight possible limitations and bottlenecks.

The presented experiments emphasize write-related workloads. This is because, first, they are the most critical for a backup appliance and, second, other operations (e.g., object deletes or MPUs) largely incur similar or lower overheads on the performance of the system.

Most of the experiments were conducted on a testbed composed of 12 servers, which, for the following reasons, was sufficient to show how our solutions behave at scale. First, the data was kept in a default 9+3 erasure-code scheme. In effect, with 12 servers or more, each machine stored only one fragment of erasure-coded data, and that would not have changed if the system had grown further. Second, considering the maximal raw capacity of one HYDRAstor server, which is 168 TB, the total capacity of a 12-server system was over 2 PB. In other words, it was a fairly large installation considering deduplication, especially given that, in contrast to HYDRAstor, many popular backup appliances available on the market do not scale more [32], [33]. Finally, the experiments put a considerable pressure on our infrastructure: altogether they took several weeks. Therefore, even in the scaling tests, we had to limit the maximal size of our testbed to 18 servers. Each of the servers comprised 2x Intel Xeon CPUs E5-2620 v3 2.40GHz, E5-2660 v3 2.60GHz, or E5-2430 2.20GHz, 96 GB of RAM, and 12x 7200-RPM SATA HDDs of 2–6 TB each.

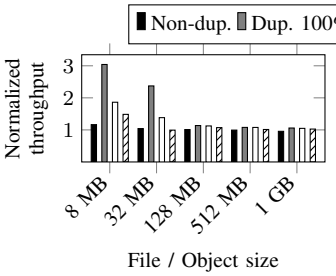

Fig. 12: ObjDedup throughput normalized to results of the file system.

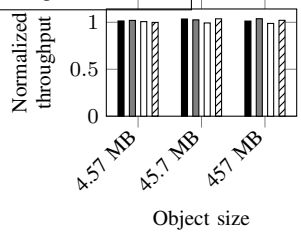

Fig. 13: Throughput of writing with three different object sizes normalized to throughput of writing with the one (average) size.

A. Assessment of the Main Performance Goals

We start by assessing the main performance goals of ObjDedup. To this end, we evaluate its backup throughput (Section VI-A1), scalability (Section VI-A2), and overheads (Section VI-A3).

1) Backup Throughput: We test ObjDedup with varying object sizes and four different workloads: 100% non-duplicate writes, 100% duplicate writes, 90%:10% duplicate:non-duplicate writes, and reads. The first two workloads are extreme scenarios but still possible in practice (e.g., writing an initial backup without internal duplicates and writing the same backup twice). The 90%:10% duplicates:non-duplicates is a workload that yields a 10:1 deduplication ratio, which is lower than the previously mentioned 20:1 ratio frequently achievable in the real world. Nevertheless, we show the results for the 90%:10% workload, because the results for the 95%:5% workload are more similar to the 100% workload. Finally, the workload with reads is for reference. Each experiment involves a 2.4-TB data set, which is large enough to get reproducible results.

Figure 12 presents the throughput of ObjDedup normalized to the results of the HYDRAstor file system driver in the same configuration, which we use as a baseline. The object/file sizes vary from 8 MB, which is rather small even for object storage backup (we present results for yet smaller files in further experiments), to 1 GB, as increasing object size has a marginal impact on performance from some point. Both drivers achieve a comparable write performance for larger object/file sizes (over 128 MB). Moreover, since ObjDedup reuses the deduplication implementation (e.g., chunking, fingerprinting), it achieves the same deduplication ratios. With smaller object/file sizes, there are differences in write performance in favor of ObjDedup. The read throughput is comparable for all object sizes; the only noticeable difference is for 8 MB objects in favor of ObjDedup.

Since ObjDedup is expected to handle simultaneous streams from multiple backup applications, the data in each of the streams can come in differently-sized objects. Nevertheless, as Fig. 13 shows, the throughput of writing data from three simultaneous streams with different object sizes (2/8/32 MB, 20/80/320 MB, 200/800/3200 MB) does not diverge significantly from the throughput of writing the same streams with a single object size (4.57/45.7/457 MB). The object size was selected as an average object size when three streams of equal

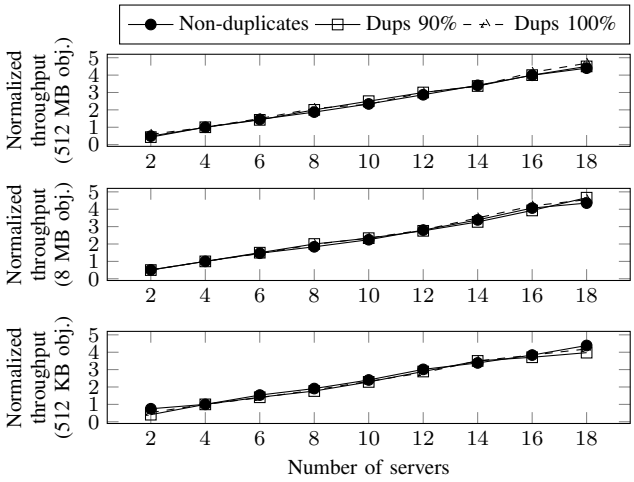

Fig. 14: Scalability of ObjDedup (normalized to results for 4 servers).

size were written with three different object sizes (e.g., when three 1024 MB streams are written in 2/8/32 MB objects, the average object size is 4.57 MB).

2) Scalability: To demonstrate the scalability of ObjDedup, Fig. 14 depicts the throughput of writes in the three different write workloads (100% non-duplicates, 100% duplicates, 90%:10% duplicates:non-duplicates) and with three different object sizes (512 KB, 512 MB, and 8 MB). The first size represents small objects, the second—to large ones, and the third is a mean between 512 KB and 128 MB, which was the level-off point in Fig. 12. The presented values are normalized to the per-server throughput of 4-server HYDRAstor setup.

On average, 18 servers were 4.41 times faster than four. The worst result (an average improvement of 4.18) was obtained with the smallest objects (512 KB), which reinforces the claim that efficiently handling small objects is not trivial. The smallest possible distributed configuration (2 servers) is also included in the plot, but the results are a bit skewed, as such configuration uses far less network communication.

3) Overheads: We also measured and evaluated the overheads incurred by ObjDedup in terms of resource consumption (memory, storage, CPU), response latency, and scaling.

Memory consumption: The major cost is buffering data incoming through the HTTP server. The buffering is required to ensure that all components of the system have enough work to achieve a high level of parallelism. In our experiments, we use a 2-GB buffer per server, which is sufficiently large. The memory overhead of other data structures is in turn significantly smaller (e.g., metadata of 50k objects in an OML take less than 200 MB, even with 3 KB of metadata per object). Overall, none of our experiments consumed more than 3 GB of RAM per server.

Storage space consumption: Data chunks are referenced, and these references can take up a considerable amount of storage space. However, this happens with any interface, not just ObjDedup. Similarly, objects contain their metadata (up to 3 KB per object), but any object storage system must keep these. Therefore, the most important aspect is quantifying additional overheads in OMT and OML.

For each object, besides its metadata, an OMT leaf keeps 20 bytes of our internal metadata. Second, there are internal OMT blocks (storing object keys of up to 1 KB and separators), but with a branching degree $S = 64$, there are on average 48x fewer of them in the penultimate level of an OMT than leaves, and far fewer on all other levels combined. Finally, an OML keeps metadata of a limited number of objects (typically ∼50k), which is negligible in comparison to the millions of objects kept in the system. What is important, however, is that, during a merge, each object in an OML can cause a rewrite of a whole OMT leaf, and both versions of such a leaf need to be stored until the merge is finished. Therefore, for S=64, an OML can store as many as 3.2 M of object metadata copies. To sum up, with a realistic workload for backup data (e.g., object data being considerably larger than object keys) and multi-terabyte storage servers, the storage capacity overheads incurred by ObjDedup are far below 1% of the system capacity.

CPU consumption: If HTTPS is enabled, the majority of CPU load is due to encryption and decryption, as providing multi-gigabyte throughput with cryptographic algorithms can require multiple cores. The CPU consumption of ObjDedup itself, in turn, highly depends on the workload. Typically, it does not exceed 3 cores per server, mostly handling HTTP requests and managing the OMT.

Latency: In the expected write-dominant backup workloads, the latency overhead is marginal and mostly comes from the fact that at the end of writing an object, an OML entry must be written in the block storage. However, there are two cases in which ObjDedup increases the latency considerably. First, when reading an object, multiple levels of the OMT are accessed, unless at least some of them are cached. In effect, the time of arrival of the first bytes of the data is proportional to the height of the OMT. For instance, if the OMT has 5 levels and a block read takes 100 ms because of the concurrent load, reading object data will take 500 ms. Second, a metadata merge can cause numerous additional I/Os per object if object keys are non-sequential, which we study in microbenchmarks (Section VI-C). In such cases, the storage cluster can become overwhelmed with requests and have several times longer response times, which ultimately increases the latency of all operations.

Scaling: As shown previously, the throughput of ObjDedup scales nearly linearly (cf. Fig. 14), just as bare HYDRAstor. However, this is true as long as metadata handling does not become a bottleneck. We study such problematic scenarios in the microbenchmarks (Section VI-C).

B. Comparison with Existing Solutions

In this section, we show how ObjDedup compares against the state of the art. As mentioned previously, we are not aware of any prior work that closely matches ours, and hence for comparison, we select systems that are close to our solutions in as many relevant aspects as possible. In Section VI-B1, we thus compare ObjDedup to two state-of-the-art open-source object stores. In Section VI-B2, in turn, we compare it to three file systems with deduplication.

Furthermore, performing the comparison was still challenging because of significant differences between ObjDedup and the reference systems. To avoid bias in favor of our solutions, the experiments evaluated ObjDedup in an unfavorable setup: in the

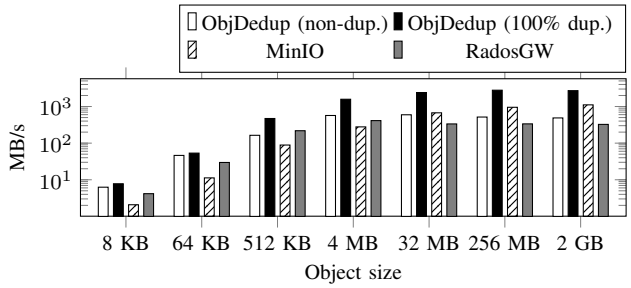

Fig. 15: Write throughput (log. y-axis) with varying object sizes.

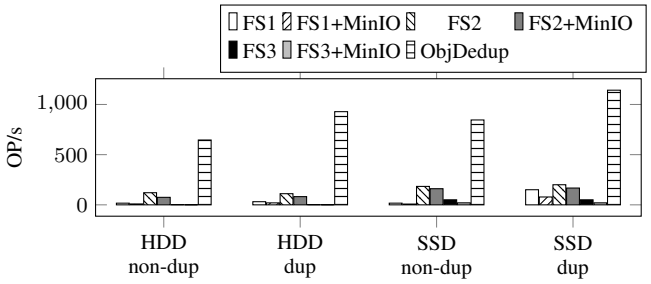

Fig. 16: Number of operations (file copy / object put) per second.

first one, ObjDedup was the only object store that performed deduplication during non-duplicate writing, and in the second, the configuration was far from what ObjDedup was designed for.

1) Backup Throughput: In the first experiment, we used COSBench [73] to compare ObjDedup with other object stores. COSBench is a framework that enables performance testing of object storage but does not support writing duplicates. Therefore, we modified its code for that purpose. As the two reference systems, we selected Ceph with RadosGW and MinIO, which are state-of-the-art open-source object stores. We chose them despite their lack of in-line deduplication, because we did not find any alternative offering this feature that could be used in the experiments or had comparable results published.

In the experiment, we used one of our servers (2x Intel Xeon CPU E5-2620 v3 2.40GHz and 12x 4 TB SATA HDD). We decided upon the single-server setting, as it significantly simplified the setup and result analysis. Each of the three systems used 9+3 erasure codes, which give a high failure resilience with 12 disks and are broadly adopted (e.g., they are proposed in Ceph's documentation [74]). To store data internally, we used XFS for MinIO and BlueStore for RadosGW, which are recommended to achieve a high performance. The object sizes in the presented experiments vary from 8 KB to 2 GB, as decreasing them even further did not affect the number of operations per second and increasing them did not affect the throughput.

For non-duplicates, ObjDedup achieves a similar throughput as the others (Fig. 15). Despite the fact that deduplication consumes additional resources when writing non-duplicates, ObjDedup is either the fastest or the second fastest system. With duplicates, in turn, it has a 1.8–3.83x higher throughput than the others (up to 2790 MB/s), which is expected because for duplicate writes in-line deduplication can overcome the limits of HDDs. Finally, in COSBench read tests (not plotted), the performance of ObjDedup is also comparable to MinIO and RadosGW. With small objects, it exceeds 740 GET/s. Its throughput with 32MB or larger objects is in turn close to 800 MB/s.

2) Files/Objects per Second: In the second experiment, we compared ObjDedup with existing deduplication solutions. Therefore, we selected three commercial file systems with in-line deduplication. We do not disclose their vendors, as our goal is to validate ObjDedup and not to compare the products.

Some of the file systems were designed for a mix of HDDs and faster storage devices (e.g., for a write-back journal). Therefore, to find the common denominator and make it possible to compare the solutions, we employed two testbeds: the first using SATA 1 TB 7200 RPM HDDs and the second using NVMe 1 TB SSDs for all kinds of storage. Both configurations were very different from what ObjDedup was tuned for, especially the full-SSD one, but facilitate result reproduction. In both of them, a single machine with Intel Core i7-7820X @ 3.60GHz and 64 GB RAM was used, as some of the file systems do not scale to more machines.

The presented experiments were conducted using MinIO's client, which can copy data to both file systems and object stores. Nevertheless, experiments with other tools gave similar results. We also conducted experiments with MinIO configured as a layer on top of each file system to provide object storage with in-line deduplication.

Initially, we intended to show results for objects and files of various sizes, similarly to the previous experiments. However, for objects below 100 MB, ObjDedup was typically one or two orders of magnitude faster, so we decided to investigate the phenomenon even further. Therefore, we limited the contents and file names to 32 bytes, despite the fact that ObjDedup was not designed to handle data in such small objects efficiently. In that way, we were able to measure the upper bound for operations (file creates or object PUTs) of each solution. The same directory was copied twice, so in the second run, all data were duplicates.

As shown in Fig. 16, ObjDedup can handle 4.6–8.35x more operations than the fastest of the file systems. In general, file systems with in-line deduplication are complex, so handling that many files per second is challenging for them. In contrast, ObjDedup applies updates to the OMT in batches, so they are highly efficient. Additionally, MinIO's client copies files to a temporary location and renames them afterward to prevent listings on inconsistent files, so two operations are needed per file. Compared with MinIO on top of a file system solution (FS+MinIO in Fig. 16), ObjDedup can handle 5.26–11.34x more PUT/s, which is justified as an additional layer introduces new overheads. To sum up, even with such a minimal object size, none of the file systems reaches a request rate comparable to ObjDedup.

C. Microbenchmarks

OMT is our novel data structure essential for high performance. If a metadata merge takes too long, new requests cannot be handled, and the system throughput is decreased.

For instance, if an OML stores up to 50k entries and a merge takes 100 seconds, the system cannot handle more than 500 PUT/s. Therefore, we evaluate the metadata merge and the distributed metadata merge in a series of experiments that show their performance under various circumstances.

In Sections VI-C1, VI-C2, and VI-C3, we study how the pattern of the workload affects the performance of a merge. The experiments were on-purpose conducted in a rather small configuration to emphasize the impact of the pattern and not system scaling. More specifically, the configuration involved two servers with 12x SATA 7200 RPM 6 TB HDDs each and 2x Intel Xeon CPU E5620 @ 2.40GHz or 2x Intel Xeon CPU E5620 @ 2.40GHz. The first server hosted both an object driver and a storage cluster node, and the second—just a storage node. Our testbed thus consisted of more than one server but only one object driver conducted merges.

For this reason, to analyze the distributed merge, in Section VI-C4, we present how a merge scales from 1 to 16 servers under two different workload patterns.

To discuss the experiments, let us explain the object key patterns they utilized. In the *rand* pattern, all keys consisted of generated UUIDs, so object metadata were uniformly distributed across a large number of OMT leaves. In the *seq* pattern, objects were written to 1000 different prefixes (50 characters of a prefix followed by subsequent integer numbers), so consecutive objects belonged to the same or neighboring leaves. *Seq* thus approximated what we would expect from backup applications, while *rand* modeled a theoretical worst case. In addition, in some experiments—those with suffix *-delay*—block reads in the storage cluster were delayed by 150 ms to simulate a system overloaded with other tasks (e.g., due to drivers other than ObjDedup). Finally, unless stated otherwise, up to 50k entries were stored in an OML before a merge was initiated.

1) Prefetch Algorithms: First, we evaluate the impact of our metadata merge prefetching (OMT Prefetch). Without any prefetching, waiting for consecutive reads increases the merge time to tens of minutes, even when an OMT contains less than a million objects (Fig. 17). Therefore, prefetching is simply a necessity.

However, with a naive approach, referred to as the AllInt Prefetch, which simply prefetches all internal nodes, the number of read nodes increases linearly with the total number of nodes, even in the *seq* pattern (Fig. 18). In contrast, with the OMT Prefetch, it stabilizes around 10 seconds.

Since merges without the OMT Prefetch are slow, all experiments in the subsequent sections thus utilize it.

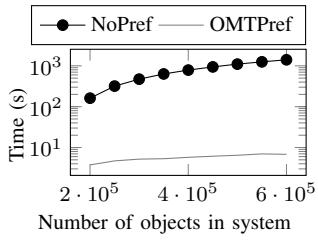

Fig. 17: OMT merge with data written in *rand* pattern.

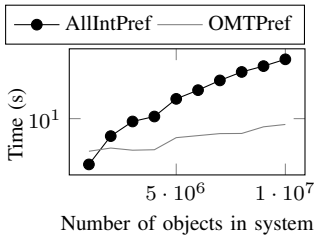

Fig. 18: OMT merge with data written in *seq-delay* pattern.

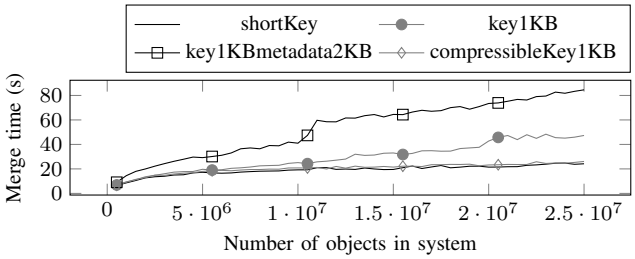

Fig. 19: Merge time depending on key and user-defined metadata size.

2) Key and Metadata Sizes: As described in Section III-A, an object key can consume 1 KB. Each OMT node stores full keys, so the key size affects the amount of information that is written and read during a merge. Object metadata, which consume additional kilobytes, are not kept in internal nodes but can be stored in leaves to decrease the number of I/Os for HEAD requests.

Long keys can increase the time of a merge up to 2x (cf. Fig. 19) if a bottleneck on throughput of processed metadata arises. Moreover, if each OMT leaf stores additional 2 KB of uncompressible metadata, the time of a merge grows even further (*key1KBmetadata2KB* in Fig. 19). In such a scenario, keeping object metadata together with object data should be considered. In practice, a block containing multiple keys can be compressed when the keys are similar (typically, long keys have a common prefix). If keys are large, but blocks holding them compress well, there is hardly any impact on the merge time (*compressibleKey1KB* in Fig. 19).

3) Number of Objects: As shown in Fig. 20, irrespective of the total number of objects in the system, for the *seq* pattern, a merge takes at most 10–11 seconds, even with an artificial delay of 150 ms on the storage cluster emulating other load. This is because under this pattern the locality is high, so most of new objects are added in groups to new OMT leaves, and also few of the internal OMT nodes require rewriting.

The results are much different with the *rand* pattern, as the changes are distributed across the whole OMT. Therefore, the top levels of the OMT are almost completely rewritten, and each of the deeper levels requires up to 50K changes. In Fig. 20, the plot for the *rand* pattern looks almost like a linear function despite the fact that tree height is a logarithm of the objects number. Two phenomena contribute to this behavior. First, the number of internal nodes is small (about 10K for 25M objects), so most of them are rewritten when 50K randomly distributed objects are added. Second, the efficiency of caching in the storage cluster diminishes, because the larger the tree is, the less data locality.

To get more insight into how the bottleneck on reads from the storage cluster affects metadata merge for the *rand* pattern, we include results for two different types of erasure codes, that is, apart from the default 9+3, also 3+9. In *rand/9+3* and *rand-delay/9+3*, each read of a block needs 9 disk accesses. With such a volume of disk read I/Os, the merge time increases quickly for over 40M objects when the cache efficiency drops. In contrast, with 3+9 codes, each read needs 3 disk accesses, so the disks were not overloaded, and a merge can finish almost twice as fast. Note that there are techniques that allow for

reducing disk accesses, such as adopting erasure codes requiring fewer I/Os during reads or simply caching a larger fraction of internal OMT nodes.

4) Distributed Metadata Merge: The last microbenchmark evaluates the distributed metadata merge in configurations of up to 16 servers hosting object drivers. In such a distributed setup, each object driver writes into its own OML, so the number of entries processed by each merge can be increased without changing the size of the OML per driver. In other words, an increased number of servers increases the number of objects processed per merge.

Figure 21 shows for the *rand* pattern how the time of a merge changes when the number of objects driver instances and storage nodes (one per driver instance) increases with the number of entries per merge. Despite the driver coordination overhead due to the distribution, 16 servers are able to merge 800k objects faster than 8 servers merge 400k objects, 4 servers merge 200k objects, or 1 server merges 50k (using a non-distributed merge). This is because, with more objects in a single merge, the ratio of changed internal OMT nodes to leaf nodes decreases. In other words, the more changes to apply in a merge, the higher the probability that two or more objects have a common leaf or internal nodes. This characteristic enables handling the same load more than twice as fast when the number of servers is doubled.

On the other hand, the total number of objects in a system with more servers could likely be larger as well. In such a scenario, there is some loss resulting from the load distribution overhead and the increased subtree heights. For instance, for 15M objects in the system with 8 servers, a merge takes 45.585 s, while for 60M objects and 16 servers, it takes 50.951 s. Again, however, it is worth emphasizing that these results are for the *rand* pattern, which entails a lot of reads dispersed across virtually all parts of the OMT.

A backup job, in contrast, typically affects only a consistent subset of the OMT (e.g., the keys have a common prefix). Therefore, we also analyze the *seq* pattern, scaling the number of prefixes for which data are written sequentially (1000 with one server, but 16,000 with 16 servers). As can be seen in Fig. 22, similarly to the previous *seq* experiments, the total number of objects has a marginal impact on the time of a merge. Moreover, as for the *rand* pattern, the merge time decreases with the number of servers. These are highly desirable behaviors for the considered backup applications.

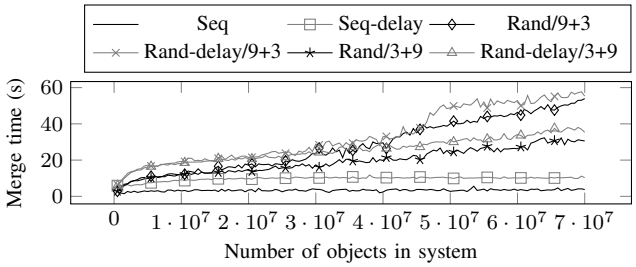

Fig. 20: OMT merge with different key naming and EC schemes.

VII. Conclusions

To sum up, there is a growing market demand for object storage interfaces in scale-out backup appliances with deduplication. Using empirical data from 686 real-world deployments of such commercial systems, we showed that a key challenge when aiming to provide support for such interfaces efficiently is the management of object metadata resulting from a different data organization and usage patterns of object storage. To address this problem, we proposed ObjDedup, a suite of distributed data structures and algorithms optimized to keep object metadata in immutable globally deduplicated block storage. We implemented ObjDedup as a layer on top of HYDRAstor and evaluated the implementation experimentally. The obtained results indicate that the performance of our solutions is comparable to that of the classic interfaces offered by HYDRAstor, despite the more challenging usage patterns. Moreover, our solutions can handle significantly more requests per second (5.26–11.34x) than object storage implementations on top of file systems provided by state-of-the-art deduplication solutions. Likewise, compared to leading object stores without in-line deduplication, it offers a much higher throughput when writing duplicate data (1.8–3.93x), not to mention the compelling storage cost reductions due to deduplication.

From a broader perspective, our preliminary study and evaluation of ObjDedup show that the trend in backup applications dedicated to object storage to write data as relatively small objects is problematic for traditional backup systems with deduplication. While ObjDedup addresses these challenges in a wide range of common configurations, we also demonstrated corner cases that are particularly hard to handle, such as extremely small objects, large keys that do not compress well, or keys without locality. As a result, we believe our study can guide backup applications in how to adjust object writing patterns to maximize performance of deduplicating storage. The presented solutions have also facilitated cloud-tiering for backup appliances [75].

References

[1] Amazon Web Services Inc., "Amazon simple storage service user guide," 2021. [Online]. Available: docs.aws.amazon.com/AmazonS3/latest/userguide/s3-userguide.pdf

[2] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, "Early observations on the performance of windows azure," in *ACM HPDC*, 2010.

[3] K. Lillaney, V. Tarasov, D. Pease, and R. Burns, "The case for dual-access file systems over object storage," in *USENIX HotStorage*, 2019.

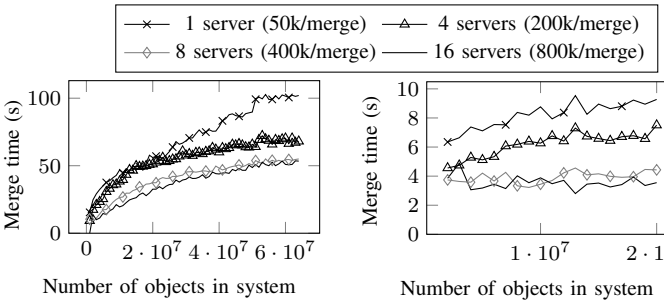

Fig. 21: OMT distributed merge time with *random* keys.

Fig. 22: OMT distributed merge time with *seq* keys.

[4] J. Barr, "Celebrate 15 years of amazon s3," 2021. [Online]. Available: aws.amazon.com/blogs/aws/amazon-s3s-15th-birthday-it-is-still-day-1-after-5475-days-100-trillion-objects/

[5] Microsoft, "Azure blob storage," 2021. [Online]. Available: azure.microsoft.com/en-us/services/storage/blobs/

[6] Alibaba, "Alibaba cloud object storage service," 2021. [Online]. Available: alibabacloud.com/product/oss

[7] Backblaze, "Backblaze b2," 2021. [Online]. Available: backblaze.com/b2/cloud-storage.html

[8] Scality, "Scality," 2021. [Online]. Available: scality.com

[9] Ceph, "Ceph object gateway," 2021. [Online]. Available: docs.ceph.com/en/latest/radosgw/

[10] OpenStack, "Openstack object storage," 2021. [Online]. Available: wiki.openstack.org/wiki/Swift

[11] Minio, "Minio object storage," 2021. [Online]. Available: min.io

[12] B. Alon, "Mezzfs," 2021. [Online]. Available: netflixtechblog.com/mezzfs-mounting-object-storage-in-netflixs-media-processing-platform-cda01c446ba

[13] Amazon Web Services Inc., "Pinterest on aws," 2021. [Online]. Available: aws.amazon.com/solutions/case-studies/innovators/pinterest/

[14] P. Matah, "Minecraft earth and azure cosmos db part 1," 2021. [Online]. Available: azure.microsoft.com/pl-pl/blog/minecraft-earth-and-azure-cosmos-db-part-1-extending-minecraft-into-our-real-world/

[15] DELL EMC, "Storage s3 in backup," 2021. [Online]. Available: dellemc.com/content/dam/uwaem/production-design-assets/pl-pl/events/forum/2017/presentations/cloud_lanscape4.pdf

[16] Bareos, "Bareos droplet storage backends," 2021. [Online]. Available: docs.bareos.org/TasksAndConcepts/StorageBackends.html

[17] CockroachLabs, "Cockroachdb backup," 2021. [Online]. Available: cockroachlabs.com/docs/dev/backup

[18] Teradata, "Teradata using amazon s3 storage as the backup target," 2021. [Online]. Available: docs.teradata.com/r/CCZ_TZJngXILEsdDOzUoAw/WOt0MU3umZEx7P7mcKH8Eg

[19] Veritas, "Veritas netbackup™ cloud administrator's guide," 2019. [Online]. Available: veritas.com/content/support/en_US/doc/58500769-135186602-0/v126619409-135186602

[20] Veeam, "Object storage repository," 2021. [Online]. Available: helpcenter.veeam.com/docs/backup/vsphere/object_storage_repository.html

[21] Dell EMC, "Optimized data protection with integrated deduplication," 2021. [Online]. Available: dell.com/downloads/global/products/pvaul/en/dell-emc-dd-series-brochure.pdf

[22] C. Dubnicki *et al.*, "HYDRAstor: A scalable secondary storage." in *USENIX FAST*, 2009.

[23] M. Oh, S. Park, J. Yoon, S. Kim, K.-w. Lee, S. Weil, H. Y. Yeom, and M. Jung, "Design of global data deduplication for a scale-out distributed storage system," in *IEEE ICDCS*, 2018.

[24] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Usenix Fast*, 2008.

[25] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM ToS*, vol. 7, no. 4, pp. 1–20, 2012.

[26] Q. Corporation, "Why quantum dxi purpose-built appliance?" 2021. [Online]. Available: cdn.allbound.com/iq-ab/2020/02/ST02281A-v01.pdf

[27] Minio, "High performance object storage for veeam backup and recovery," 2020. [Online]. Available: blog.min.io/minio-high-performance-object-storage-for-veeam-backup-and-recovery/

[28] Commvault, "Public cloud architecture guide for microsoft azure," 2020. [Online]. Available: documentation.commvault.com/commvault/v11/others/pdf/public-cloud-architecture-guide-for-microsoft-azure11-19.pdf

[29] G. Wallace *et al.*, "Characteristics of backup workloads in production systems." in *USENIX FAST*, 2012.

[30] J. Paulo and J. Pereira, "Efficient deduplication in a distributed primary storage infrastructure," *ACM ToS*, 2016.

[31] H. Wu *et al.*, "Hpdedup: A hybrid prioritized data deduplication mechanism for primary storage in the cloud," *arXiv:1702.08153*, 2017.

[32] Quantum, "Dxi-series backup appliances," 2021. [Online]. Available: cdn.allbound.com/iq-ab/2021/04/DXi-DS00549A.pdf

[33] DELL EMC, "Data domain deduplication storage systems," 2018. [Online]. Available: delltechnologies.com/asset/en-us/products/data-protection/technical-support/h11340-datadomain-ss.pdf

[34] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM CSUR*, 2014.

[35] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[36] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, "Anchor-driven subchunk deduplication," in *SYSTOR*, 2011.

[37] Z. J. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "Cluster and single-node analysis of long-term deduplication patterns," *ACM Transactions on Storage (TOS)*, 2018.

[38] J. Wei, H. Jiang, K. Zhou, and D. Feng, "Mad2: A scalable high-throughput exact deduplication approach for network backup services," in *IEEE MSST*, 2010.

[39] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *USENIX ATC*, 2010.

[40] Y. Allu, F. Douglis, M. Kamat, R. Prabhakar, P. Shilane, and R. Ugale, "Can't we all get along? redesigning protection storage for modern workloads," in *USENIX ATC*, 2018.

[41] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *SYSTOR*, 2012.

[42] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *USENIX FAST*, 2013.

[43] Z. Cao, H. Wen, F. Wu, and D. H. Du, "Alacc: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching," in *USENIX FAST*, 2018.

[44] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. Du, "Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance," in *USENIX FAST*, 2019.

[45] P. Strzelczak *et al.*, "Concurrent deletion in a distributed content-addressable storage system with global deduplication," in *USENIX FAST*, 2013.

[46] F. Douglis *et al.*, "The logic of physical garbage collection in deduplicating storage," in *USENIX FAST*, 2017.

[47] OpenStack, "S3/swift rest api comparison matrix," 2021. [Online]. Available: docs.openstack.org/swift/latest/s3_compat.html

[48] Google Cloud, "Migrating from amazon s3 to cloud storage," 2021. [Online]. Available: cloud.google.com/storage/docs/migrating

[49] OpenStack, "Configure object storage with the s3 api," 2021. [Online]. Available: docs.openstack.org/mitaka/config-reference/object-storage/configure-s3.html

[50] A. Duggal *et al.*, "Data domain cloud tier: Backup here, backup there, deduplicated everywhere!" in *USENIX ATC*, 2019.

[51] G. Cheng, D. Guo, L. Luo, J. Xia, and S. Gu, "Lofs: A lightweight online file storage strategy for effective data deduplication at network edge," *IEEE TPDS*, 2021.

[52] J. Wang *et al.*, "Towards cluster-wide deduplication based on ceph," in *IEEE NAS*, 2019.

[53] A. Khan, C.-G. Lee, P. Hamandawana, S. Park, and Y. Kim, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," in *IEEE MASCOTS*, 2018.

[54] A. Khan, P. Hamandawana, and Y. Kim, "A content fingerprint-based cluster-wide inline deduplication for shared-nothing storage systems," *IEEE Access*, vol. 8, pp. 209 163–209 180, 2020.

[55] P. Hamandawana *et al.*, "Crocus: Enabling computing resource orchestration for inline cluster-wide deduplication on scalable storage systems," *IEEE TPDS*, 2020.

[56] Ceph, "Rados gateway data layout," 2021. [Online]. Available: docs.ceph.com/en/latest/radosgw/layout/

[57] J. Ma, G. Wang, and X. Liu, "Dedupeswift: object-oriented storage system based on data deduplication," in *IEEE Trustcom/BigDataSE/ISPA*, 2016.

[58] G. Cloud, "Request rate and access distribution guidelines," 2021. [Online]. Available: cloud.google.com/storage/docs/request-rate

[59] Arcserve, "Arcserve udp 8.0 is now available," 2021. [Online]. Available: support.arcserve.com/s/article/Arcserve-UDP-8-0-Is-Now-Available

[60] Nakivo, "Ransomware protection with nakivo backup & replication," 2021. [Online]. Available: nakivo.com/ransomware-protection/

[61] M. Bose, "Data protection fundamentals: How to backup an amazon s3 bucket," 2021. [Online]. Available: nakivo.com/blog/how-to-backup-an-amazon-s3-bucket/

[62] Veeam, "Define job schedule," 2022. [Online]. Available: helpcenter.veeam.com/docs/backup/vsphere/backup_job_schedule_vm.html

[63] G. Amvrosiadis and M. Bhadkamkar, "Getting back up: Understanding how enterprise data backups fail," in *USENIX ATC*, 2018.

[64] Veeam, "Short-term retention policy," 2022. [Online]. Available: helpcenter.veeam.com/docs/backup/vsphere/backup_copy_simple_retention.html

[65] IDC, "Data creation and replication will grow at a faster rate than installed storage capacity," 2021. [Online]. Available: idc.com/getdoc.jsp?containerId=prUS47560321

[66] G. Graefe and H. Kuno, "Modern b-tree techniques," in *IEEE ICDE*, 2011.

[67] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[68] M. Athanassoulis and S. Idreos, "Design tradeoffs of data access methods," in *SIGMOD*, 2016.
[69] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE TKDE*, 2015.
[70] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *USENIX FAST*, 2016.
[71] Q. Kang *et al.*, "Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation," *IEEE TPDS*, 2020.
[72] G. Singh, "Leader election in the presence of link failures," *IEEE TPDS*, 1996.
[73] Q. Zheng, H. Chen, Y. Wang, J. Zhang, and J. Duan, "Cosbench: Cloud object storage benchmark," in *ACM/SPEC ICME*, 2013.
[74] Ceph, "Erasure coded placement groups." [Online]. Available: docs.ceph.com/en/mimic/dev/osd_internals/erasure_coding/
[75] I. Kotlarska, A. Jackowski, K. Lichota, M. Welnicki, C. Dubnicki, and K. Iwanicki, "InftyDedup: Scalable and cost-effective cloud tiering with deduplication," in *USENIX FAST*, 2023.

Andrzej Jackowski is PhD student at the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw. He received BSc and MSc degrees in computer science from the University of Warsaw in 2015 and 2017. He teaches courses covering distributed systems, cloud computing and operating system internals. Currently, he leads an R&D team developing HYDRAstor in 9livesdata. In 2016 he worked in the cloud computing field in Microsoft, WA, Redmond.

Łukasz Ślusarczyk received his MSc degree in computer science and in mathematics from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 2004 and in 2005 respectively. He works in data storage industry as a technical team leader. Till 2022 he was developing distributed space reclamation algorithms, replication, and object storage interface for HYDRAstor. Currently, he works upon Persistent Memory in Intel.

Krzysztof Lichota received his MSc degree in computer science and in mathematics from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 2002. He is the Senior Technical Expert at 9livesdata in HYDRAstor backend. Prior to that he worked on storage technologies for NEC Laboratories America and StorageNetworks.

Michał Wełnicki received his MSc degree in computer science and from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 2005. He is a Senior Technical Expert at 9livesdata in HYDRAstor backend. Prior to that he worked on storage technologies for NEC Laboratories America.

Rafał Wijata received his MSc degree in computer science from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 2001. He is the Senior Software Developer at 9livesdata in HYDRAstor backend. His experience encompasses engineering low-level persistence layers and file systems for distributed storage.

Mateusz Kielar received his MSc degree in computer science from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 2011. He is the Senior Software Developer developing object storage interface and WAN replication for HYDRAstor. His interests include object oriented programming and distributed systems.

Tadeusz Kopeć Tadeusz Kopeć received his MSc degree in computer science from the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw in 1999. He is the Technical Leader of team developing object storage interface for HYDRAstor. Besides distributed storage and operating systems, his interests include logic and theory of computations.

Cezary Dubnicki leads 9livesdata, a company specializing in advanced storage systems R&D. Prior to founding 9livesdata, Cezary was a department head in NEC Lboratories, Princeton, NJ and a researcher in Princeton University. Cezary obtained his PhD in CS from University of Rochester and MSc from the University of Warsaw. His research interests concentrate in systems,including storage, networking, operating systems and distributed systems.

Konrad Iwanicki is Associate Professor at the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw, where he leads the computer systems research group. He obtained his PhD from the VU Amsterdam. His research interests include distributed systems and network protocols. Apart from storage systems, recently he has been working mainly on low-power wireless networking for dependable IoT systems as well as exploring novel frontiers of cyber-physical technologies.