Smash: Flexible, Fast, and Resource-efficient Placement and Lookup of Distributed Storage

YI LIU, SHOUQIAN SHI, MINGHAO XIE, HEINER LITZ, and CHEN QIAN, University of California, Santa Cruz, United States

Large-scale distributed storage systems, such as object stores, usually apply hashing-based placement and lookup methods to achieve scalability and resource efficiency. However, when object locations are determined by hash values, placement becomes inflexible, failing to optimize or satisfy application requirements such as load balance, failure tolerance, parallelism, and network/system performance. This work presents a novel solution to achieve the best of two worlds: flexibility while maintaining cost-effectiveness and scalability. The proposed method `Smash` is an object placement and lookup method that achieves full placement flexibility, balanced load, low resource cost, and short latency. `Smash` utilizes a recent space-efficient data structure and applies it to object-location lookups. We implement `Smash` as a prototype system and evaluate it in a public cloud. The analysis and experimental results show that `Smash` achieves full placement flexibility, fast storage operations, fast recovery from node dynamics, and lower DRAM cost (<60%) compared to existing hash-based solutions such as Ceph and MapX.

CCS Concepts: • **Networks** → **Storage area networks**.

Additional Key Words and Phrases: Distributed storage; Indexing algorithm; Ludo hashing

ACM Reference Format:
Yi Liu, Shouqian Shi, Minghao Xie, Heiner Litz, and Chen Qian. 2023. Smash: Flexible, Fast, and Resource-efficient Placement and Lookup of Distributed Storage. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 33 (June 2023), 22 pages. https://doi.org/10.1145/3589977

1 INTRODUCTION

Distributed storage systems, such as object stores, are widely used today to manage large-scale data in a variety of applications, including cloud computing [23, 33, 45], social networks [17], data analytics [24], and serverless computing [11]. In such a system, each data file consists of one or more named objects that are stored in a storage cluster. Each object is uniquely identified by a bit string, called as an identifier (ID), name, or key. This paper studies object storage systems in particular but the methods proposed in this work can be used for general distributed storage.

Object placement and lookup represent fundamental tasks that need to be provided by any storage system. To manage objects on a massive scale, there are two typical object placement and lookup strategies. 1) Naïve directory-based approaches as shown in Fig. 1(a) that store ID-location mappings in a central directory server or metadata server. Clients receive object locations by querying the server. However, in large-scale object storage, the DRAM resources needed to be spent for housing the directory are significant. For instance, storing 100 billion ID-location mappings requires > 4TB DRAM, where the majority is used to store IDs, as in practice, the average size of IDs is tens of bytes such as 16 bytes in Ceph [1] and 40 bytes in Twitter [46] or Facebook [9].

Authors' address: Yi Liu, yliu634@ucsc.edu; Shouqian Shi, sshi27@ucsc.edu; Minghao Xie, mhxie@ucsc.edu; Heiner Litz, hlitz@ucsc.edu; Chen Qian, cqian12@ucsc.edu, University of California, Santa Cruz, Santa Cruz, California, United States.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
2476-1249/2023/6-ART33 $15.00
https://doi.org/10.1145/3589977

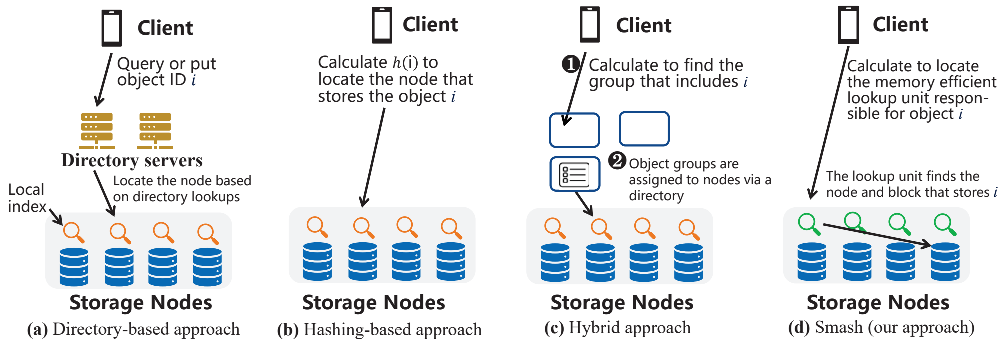

Client
Query or put object ID i
Directory servers
Locate the node based on directory lookups
Local index
Storage Nodes
(a) Directory-based approach
Client
Calculate $h(i)$ to locate the node that stores the object i
Storage Nodes
(b) Hashing-based approach
Client
❶ Calculate to find the group that includes i
❷ Object groups are assigned to nodes via a directory
Storage Nodes
(c) Hybrid approach
Client
Calculate to locate the memory efficient lookup unit responsible for object i
The lookup unit finds the node and block that stores i
Storage Nodes
(d) Smash (our approach)

Fig. 1. Comparison of different types of placement and lookup approaches for object storage.

	Storage type	Full placement flexibility	Lookup units	RAM cost	Get latency
Amazon S3 [33]	Object storage	No	Directory & local index	High	High
CRUSH (Ceph) [1, 43]	Object storage	No	Local index & OSD daemon	Low	Low
MapX [40]	Object storage	No	Local index & OSD daemon	Low	Low
HDFS [14]	File system (block-based)	No	Local index & NameNode	High	Medium
IndexFS [34]	File system	Yes	Directory & MDS index	High	Medium
InfiniFS [30]	File system	Yes	Directory & MDS index	High	Low
Smash **(this work)**	**Distributed storage**	**Yes**	**Lookup units on nodes**	**<CRUSH/MapX**	**Low**

Table 1. Comparison of typical existing data placement methods

For many applications, the data/ID space ratio is smaller than 10 for most data [46] and, as a result, many object storage systems choose an alternative option to implement object lookup: 2) Hashing-based approaches as shown in Fig. 1(b) place data to storage nodes based on the hash value of the ID $h(ID)$ [11, 22, 38, 43]. Hashing avoids the overhead of a directory but introduces several well-known issues, such as losing the flexibility of placing objects based on application requirements. The problems include placing replicas into the same failure domain, introducing load imbalance, and forcing data re-location when nodes join or leave the system [43]. A common technique to address the above problems is to use hybrid approaches (Fig. 1(c)), such as Ceph [1, 43], to find a balance between the DRAM cost of the directory and inflexibility by hashing. However they cannot completely eliminate their disadvantages. As shown in Fig. 1(c), all objects are mapped to groups using hash computations, and then all groups are assigned to different nodes via directories indicated by CRUSH [43]. Hence the grouping is still inflexible. For example, if some objects are hashed to the same group. They must be placed on the same node. When they are all popular objects, the node might receive many requests and become overloaded.

Smash addresses the challenges above by introducing a solution that achieves all desired properties of an object placement and lookup method. In particular, it enables **full placement flexibility, balanced load, low resource cost, and short latency**. As shown in Fig. 1(d), a key innovation of Smash is to apply a new space-efficient hash table and divide it into a number of distributed *lookup units*, each of which is located within an individual storage node. Each lookup unit is responsible for a group of objects whose locations can be distributed among arbitrary storage nodes. By querying an object ID, a lookup unit returns the object location, including the node and physical block address. A lookup unit can be considered a ‘shard’ of functions, including a global directory (to locate the node for an object) and local indices (to locate the block address on a node), but it achieves high resource efficiency by avoiding storing the IDs of the objects. Hence it can fit into the limited memory resource of storage nodes. For example, lookup units only cost 1.6GB DRAM per node for a storage system with 100 million objects per node, while other methods cost more than 5GB

DRAM per node. Hence Smash naturally scale out with the size of storage systems – when a system includes more nodes, it also supports more lookup units based on the configuration for Smash.

Lookup units of Smash are developed based on a recently proposed data structure Ludo hashing [37], whose theoretical basis is dynamic minimal perfect hashing (MPH) [13]. MPH significantly reduces space cost compared to a standard hash table because it avoids storing keys. Ludo and MPH is a good match to the large-volume and long-ID features of object storage because it can save most memory costs by avoiding storing IDs. Smash is **the first to apply MPH to storage systems** by dividing the whole Ludo data structure into multiple independent lookup units, which scale out with the system size. Another innovation in Smash is to decouple the metadata functions from the lookup unit and move it to *maintenance units*. Different from a central directory, the maintenance units are resource-efficient because most of them are stored on secondary storage such as SSDs and not in DRAM. Only one maintenance unit needs to be active at a time and hence needs to be resident DRAM. The active maintenance unit is responsible for handling new items that are being inserted into the storage system, whereas de-activated maintenance units governing already placed objects only need to be updated when a large amount of objects are relocated, which happens very rarely in Smash. One key insight of Smash is that insertions/deletions can be separated from lookups into two independent units. By decentralizing lookups, Smash enables high scalability and efficiency, avoiding the need for a large, centralized directory and a single point of failure.

We implement Smash and deploy it in a public cloud platform, CloudLab [8]. We evaluate the performance of Smash by comparing it with both a well-known method CRUSH [43], the placement algorithm of Ceph [1], and very recent work MapX [40], under different workloads. We show that Smash **can achieve full placement flexibility, reduce the DRAM cost per node by** $> 60\%$ **compared to other solutions, achieves very short convergence time for adding/removing nodes, and achieves low latency of putting/modifying/deleting objects.**

We show the qualitative comparison among different data placement and lookup methods in Table 1. From the table, we can find there is an obvious dilemma as directory-based methods introduce high DRAM costs while the hashing methods do not provide full placement flexibility. Our work Smash achieves the best of these two worlds: full flexibility and low DRAM cost, using a new design.

Our contributions can be summarized as follows.

- We are the first to apply MPH to storage research, and Smash resolves the flexibility-efficiency dilemma. It is the first to achieve both full placement flexibility and low resource cost, compared to the state-of-the-art object storage solutions CRUSH/Ceph and MapX. It costs less than 100MB DRAM per node for up to 6 million objects per node – >60% reduction compared to CRUSH/Ceph.
- We implement a prototype of Smash and develop it in a public cloud for evaluation. The results show that Smash achieves low latency in put/get/modify/delete operations and smaller per-node DRAM cost compared to existing object stores. Smash can also benefit from flexible object placement, such as reducing inter-rack traffic in a data center. The code of Smash is available at [7].

The rest of this paper is organized as follows. Section 2 introduces the design objectives and algorithm foundations and Section 3 presents the idea of apply MPH to storage. Section 4 describes the detailed design of Smash. Section 5 presents the prototype implementation and performance evaluation of Smash. We present the related work in Section 6 and conclude this work in Section 7.

2 OBJECTIVES AND ALGORITHM FOUNDATIONS

This section introduces the design objectives of Smash and the background of the algorithm.

# Objects per node	1M	2M	4M	8M
Bandwidth cost (TB)	9.9	19.7	39.5	78.9

Table 2. Bandwidth cost for adding one node in CRUSH [43].

2.1 Design objectives of `Smash`

We consider a large-scale object storage system, in which each object is uniquely identified by its ID. The objects are stored in storage servers called *nodes*, which may contain one or more storage devices such as SSD or HDD. Each node also carries some limited computation, DRAM, and network resources. A *block* is a sequence of bytes on a node that is read or written at a time. The *storage location* of an object i can thus be specified as $< N_i, B_i >$ where N_i is the node's network address and B_i is the sequence number of the block that stores the object. Following Ceph, the block size in `Smash` is 4MB, however, the size can be configured freely. Each block may store one or more objects. If a file is larger than 4MB, it is split into multiple objects. When a node writes objects to its disk, it keeps writing objects to a block until the block is full. Each block contains a header including the IDs of its objects and their location offsets. The objects are stored from the end of each block so that the header and objects can grow toward the middle. Clients are authorized users to access the objects, who may or may not be in the same cluster of the storage system.

The design objectives of `Smash` include: **1) Full placement flexibility**. `Smash` must support the placement of objects to arbitrary nodes, based on the application requirements for implementing fault tolerance, load balancing, data locality, and exploiting parallelism. **2) Low DRAM cost**. `Smash` needs to minimize metadata storage overheads and DRAM footprint to provide a low total cost of ownership (TCO). **3) Low latency** `Smash` needs to perform object operations such as `put`, `get`, and `delete` objects as well as adding and removing storage nodes with low latency. **4) High scalability**. When the system size increases, the extra resources to support object placement and the latency to perform lookup should increase at most linearly. To our knowledge, there is no prior work that can achieve all of these goals.

Placement flexibility is important. There are various data placement requirements of storage systems, depending on the applications of these data and the priorities of placement policies. We just name a few here. **1)** Failure tolerance. Some applications may require the replicas of some data to be in different failure domains to improve system robustness. **2)** Parallelism. Some applications may require the objects belonging to a big file or a set of files to be stored at different servers to improve accessing parallelism. **3)** Load balance. Placing data in different nodes such that no node is overwhelmed by requests for popular data is an important task in storage systems [43]. This requirement is particularly crucial for nodes with heterogeneous capacities and speeds because slow devices will become the bottleneck of overall performance. **4)** Some workloads require special placement of data to optimize performance, such as those of high-performance computing [12] and machine learning tasks [24]. Hash-based and hybrid methods cannot enforce flexible placement and hence fail to guarantee fault tolerance. They also cause further problems such as a high bandwidth cost for data migration under node addition and removal. Table 2 shows the high bandwidth cost of adding one node in CRUSH – from 10TB to 80TB – causing traffic spikes while introducing hardware costs and network overprovisioning. Our results are consistent to the reported results in [40]. `Smash` **is able to take any object-to-node placement as the input and does not need data migration under node dynamics.**

RAM cost and scalability. Every storage system requires DRAM space to support data placement and lookups. Even in hashing-based methods, where clients use hash functions to compute object locations, DRAM resource is still necessary on every node for local indices to support ID to block address mappings. The proportion of space cost to store the IDs (or keys in some context) usually contributes to the majority of the DRAM cost, e.g., $> 80\%$ [37]. The reason is that the sizes

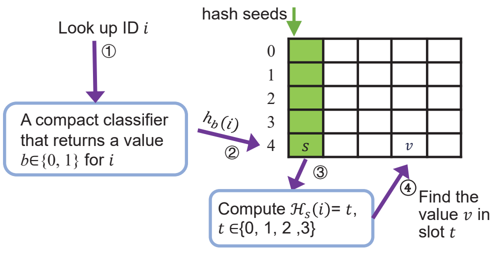

Look up ID i
①
A compact classifier that returns a value $b \in \{0,1\}$ for i
$h_b(i)$
②
hash seeds
0
1
2
3
4
s
v
③
Compute $\mathcal{H}_s(i) = t$, $t \in \{0, 1, 2, 3\}$
④
Find the value v in slot t

Fig. 2. The lookup structure of Ludo hashing.

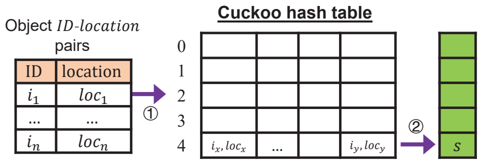


Fig. 3. The maintenance structure of Ludo hashing.

of IDs are usually much longer than those of locations – Ceph [42] uses 16-byte IDs, and Twitter's average key length is around 40 bytes [46]. It could cost hundreds of GBs DRAM for 10 billion ID-location mappings, and the majority proportion is used to store the IDs. For the methods that utilize directories, the directory and metadata servers introduce large DRAM overheads that need to be hosted by specific servers to support client queries and data management. Smash requires a number of maintenance units that can be run on either a server or storage nodes. Different from a large directory, most maintenance units can be stored on SSD because they are rarely queried or changed. Only one maintenance unit (a few hundred MBs) needs to be run in DRAM. Hence the DRAM cost of Smash is scalable to support an extremely large number of storage objects.

2.2 Ludo hashing

Allowing fully flexible placement means an object can be placed onto an arbitrary node. Hence it is essential that the system should support querying the locations of arbitrary objects from clients and remember all object-location mappings in DRAM for fast response. The biggest challenge is the DRAM cost to store the massive number of ID-to-location mappings. There is no way to avoid storing locations because they are necessary results for lookups. However, we argue that there is a way to avoid storing IDs, which contribute to a majority of the memory cost of storing ID-to-location mappings.

To enable flexibility while minimizing storage overheads, we utilize Ludo hashing [37] and adapt it for serving large-scale storage systems. Ludo is not a hash function, but a key-value lookup engine: For any given key-value mapping, Ludo can build a space-compact data structure (called the lookup structure) to return the corresponding values when keys are queried. The Ludo lookup structure does not store the keys themselves and reduces the space cost by up to 90%, compared to state-of-the-art hash tables such as (4,2)-Cuckoo [26].

The lookup structure. The lookup structure returns the value given a key – in our context key-value is ID-location. All key-value mappings are specified by the user when constructing the lookup structure and there is no restriction on the key-to-value mapping. As shown in Fig. 2, the lookup structure consists of two parts. The first stage is a classifier that returns a 1-bit value $b \in \{0, 1\}$ by querying the key i. The data structure of the classifier is called Bloomier filter [47]. Ludo selects one of two hash functions $h_0()$ and $h_1()$ based on the result of Bloomier filter b, where $b = 0$ or 1 because we have two independent hashing functions. The result $h_b(i)$ maps to a bucket (row) of a table shown on the right of Fig. 2. The bucket includes a seed value s and four slots and four elements are hashed into these four slots without collision by a seed computed by brute force. Ludo computes a hash function by including the seed value s and i, $\mathcal{H}_s(i)$, to produce a 2-bit result ranging from 0 to 3. The slot with the resulting number will be chosen and the value stored in the slot is the returned value – the object location in our context. The lookup structure is very compact and only introduces a cost of $3.76 + 1.05l$ bit per key-value pair where l is the size of each value. The query time complexity is $O(1)$.

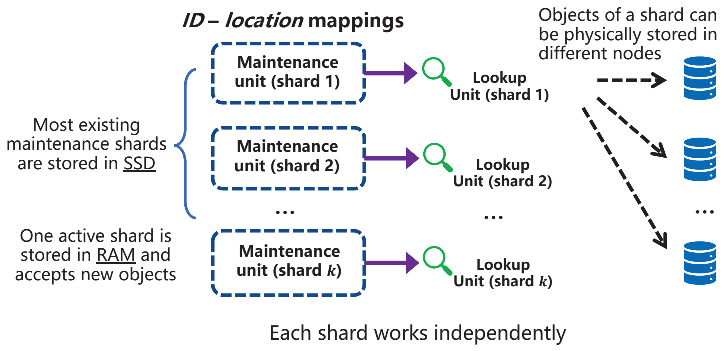

ID – location mappings
Objects of a shard can be physically stored in different nodes
Maintenance unit (shard 1)
Lookup Unit (shard 1)
Most existing maintenance shards are stored in SSD
Maintenance unit (shard 2)
Lookup Unit (shard 2)
One active shard is stored in RAM and accepts new objects
Maintenance unit (shard *k*)
Lookup Unit (shard *k*)
Each shard works independently

Fig. 4. Shards of Smash

The maintenance structure. The maintenance structure is used to construct the lookup structure. As shown in Fig. 3, it uses a (4,2)-Cuckoo hash table [26] to store all key-value mappings. Each key-value pair is stored in one of the two buckets determined by the hash result of $h_0(i_x)$ and $h_1(i_x)$. Each bucket contains four slots and a pair is stored in one of them. For each bucket, Ludo finds a seed s using brute force such that all results of $\mathcal{H}_s(i)$ for the four keys in the bucket are different. Each seed is 5-bit long and a very small portion of seeds longer than 5-bit are addressed in an overflowing table. The second stage of the lookup structure is a copy of this table with all keys being removed. Using a seed for each bucket is the key idea to perform lookups in the table without storing the keys, because the seed guarantees that there is no collision for IDs that are mapped to the same bucket. The first stage of the lookup table is a classifier that maps each key to either 0 or 1, depending on which bucket each key is stored in. The time complexity of constructing a lookup structure is $O(n)$ for n items *in expectation* and that of inserting, deleting, and changing one item is $O(1)$ *in expectation.*

3 APPLYING LUDO TO OBJECT STORAGE

Using Ludo directly as a solution for the central directory could be an improvement over existing directories, but it still suffers from some crucial problems – the lookup structure still could be a bandwidth bottleneck and the maintenance structure still introduces high DRAM space. Hence we propose two modifications when applying Ludo to Smash: 1) We divide both the lookup and maintenance structures into shards, called *lookup units* and *maintenance units*. Each shard works independently. Sharding allows most maintenance units to be stored in SSD instead of DRAM and lookup units can run on different storage nodes *in a distributed manner*. 2) We combine lookup units with the local indices of storage nodes to further reduce DRAM cost.

Sharding. Sharding the lookup and maintenance structures of Ludo is the key idea for achieving low memory cost and high scalability in Smash. Unlike existing hashing-based storage solutions that make shards (groups) of the objects and put them onto different nodes, Smash **makes shards of the lookup and maintenance structures**. Sharding the lookup and maintenance structures preserves full placement flexibility – because the locations of the objects do not affect how these structures are built. As shown in Fig. 4, all objects are divided into multiple shards. Each shard includes the objects put to the system within a period of time.

In Smash, each maintenance structure is responsible for one shard that includes a fixed number (e.g., 40 million) of objects. Objects are assigned to maintenance structures in a time-series order. For example, the first 40 million objects will be added by the first shard. Then Smash stops adding objects to the first shard, starts the second shard, and adds future objects to the second one. When a shard is full, it becomes an immutable shard and cannot be added with more objects. Deleting objects is allowed for an immutable shard. When an object is put by a user, the ID of the object is assigned by a maintenance unit running on a monitoring server, similar to CephFS [1], in which metadata servers will give each object a unique ID. In Smash, each object ID's prefix is its shard ID. For example, if the length of each shard ID is 3 bytes and an object belongs to the first shard, the object ID starts with 23 zeros and a one. Hence the shard ID of an object is directly accessible for users by looking at the object ID prefix. There is no need to keep a data structure to look up the responsible maintenance unit of an object. Such design supports a wide range of applications as long as the stored data of these applications increase gradually with time without extensive deletion operations (e.g., more than 50% of objects will be deleted in a short time), which could make each shard less efficient in maintaining lookup units. Possible applications include 1) web applications that store user activity data; 2) IoT applications that store the sensing and monitoring results from IoT devices; 3) log data of large systems. In fact, embedding an object's creation timestamp is very common in databases. For example in MongoDB [5], each object ID is a 12-byte value that includes a 4-byte timestamp that represents the number of seconds since the Unix epoch. The shard ID can be considered a very coarse-grained timestamp.

Each shard has an independent maintenance unit to construct an independent lookup unit. Each lookup unit is very compact (e.g., < 100MB) and physically stored in an arbitrary node's DRAM. An object of the shard can be physically stored in any node. When a shard is full, the maintenance unit is switched into inactive mode and stored in the SSD of the server. At this time, the lookup unit is considered to be immutable, and hence its maintenance unit is no longer needed in DRAM. Inactive maintenance units are only used in rare situations (discussed later), hence it can tolerate the latency of loading from the SSD. A new maintenance unit is started to accept new objects. The overall DRAM cost of maintenance units is small (< 250MB) even for 8 million objects per node. The number of lookup units increases with the system size, but it does not incur extra overhead because the number of nodes that can host the lookup units also increases.

Combining local indices. Most existing methods return the node's IP address as the location of an object i. Hence it is still necessary to have local indices running on nodes to return the *ID-block* mapping – a block is a basic storage unit for each read or write operation and it may store one or more objects. We find that combining the lookup units and local indices can bring tremendous savings to the DRAM cost of Smash. The location loc_i of object i returned by the lookup unit can contain both the node IP and block address: $loc_i =< IP_i, block_i >$. Our evaluation results show that such a combination reduces > 60% DRAM per node compared to the local indices of CRUSH [1], because Smash's lookup units do not store IDs. Note the IP address can be implemented as a node ID that is much shorter than 32 bits, if there is another table maintaining the NodeID-to-IP mappings or the node ID can simply be the suffix of an IP. For ease of presentation, we still use IP_i but in practice, it does not need 32 bits.

4 DESIGN OF Smash

4.1 System overview

Smash consists of three main software components: a monitor, a number of maintenance units (an active one running in the DRAM and others stored in SSD), and a number of lookup units that are located in the DRAM of storage nodes. Smash separates the tasks required to manage and look up objects into these three components to enable flexible placement without requiring a large directory. In particular, the directory-less, decentralized lookup units find objects without storing the keys that contribute to the majority DRAM cost. Lookup units provide flexibility because objects can be stored at arbitrary nodes. We leverage Ludo to scale MPH to a large number of objects per node. The task of the maintenance units is to perform updating the lookup units. In most cases, only one maintenance unit should be run in DRAM and the others can be stored in SSD. The whole system costs very little resource: it requires one server running the monitor and active maintenance unit

Term/Notation	Explanation	Example values
Node	A physical storage server	
n	Number of storage nodes	1K~100K
Monitor	A program to monitor storage space	Only one in the whole system
Shard	A virtual group of objects. Shards are not tied to nodes, and a node can host objects belonging to different shards.	1M~10M objects per shard
Maintenance unit	A program to manage a shard.	Only one is active in the whole system
Lookup unit	A program to support lookups in a shard. Can run on any server in the system. Smash co-locates them on the nodes.	One per shard.
k	Number of shards in the system. Also equal to # of maintenance units	1K~1M
α	Number of objects per shard	1M~1B
p	Number of blocks per node on average. Each block is 4MB and stores one or more objects.	1M~10M
l	Length of an object ID (key)	40Bytes
l_s	Length of a shard ID	3~4Bytes
l_n	Length of a node ID	3~4Bytes

Table 3. Terms and notations used in the paper.

and storing the other maintenance units. The lookup units are running in the DRAM of the storage nodes.

Table 3 shows the list of parameters and their explanations. Suppose a large storage system includes $n = 10$ thousand nodes, $k = 10$ thousand shards, each shard includes $\alpha = 40$ million objects. Based on our analysis (presented later), the monitor costs 391MB, a maintenance unit costs 1.5GB, and a lookup unit costs 680MB. In this setting, a server with 4GB DRAM is sufficient to run the monitor and the maintenance units and each node only needs <1GB DRAM to run the lookup units.

Monitor. The monitor provides the following functions. 1) It maintains the disk availability on all nodes at a *coarse-grained* level. Each node's space is divided into *bulks*, each consisting of 1GB data. A bulk is further divided into blocks and each block is 4MB. A block is used to store one or more objects. These sizes may vary for different applications. The bulk-level management enables the monitor to track whether each bulk has been assigned to an existing maintenance unit while it does not track whether a block has been used or not. The block-level management is performed by each maintenance unit. Hence the monitor maintains a bitmap containing one bit for each bulk in the system and each bit representing whether a bulk has been assigned. 2) The monitor tracks the resource load of every node, including disk space, network bandwidth, DRAM, and CPU. The granularity of these loads is user-specified. 3) The monitor includes a load balancing function, which can stop assigning bulks to nodes that are about to reach a high load. Therefore, it receives information about the top-k most popular objects from high-load nodes enabling load-balancing of the most frequently accessed objects among nodes. Any existing load-balancing algorithm such as [44] is compatible with `Smash` because the placement is fully flexible.

Maintenance units. Maintenance units are responsible for constructing and updating lookup units, as well as for providing fine-grained storage resource management at the *block level*. Each maintenance unit is responsible for a limited number of objects, and the set of these objects is called a *shard*. When Smash enables a maintenance unit, it is resident in DRAM of one of the servers accepting new put requests. Get requests are directly served by the lookup units and require no maintenance unit interaction and Delete requests do not need to be processed by the maintenance unit. The maintenance unit determines the object placement location (storage node) by optimizing the application requirements considering fault tolerance regions, load balancing goals, parallelism opportunities, and workload-specific requirements. The detailed optimizing algorithm is out of the scope of this paper. However, Smash can be adapted to support any algorithm. Next, the maintenance unit determines the bulk to house the object on the particular storage node. Therefore, it either reuses an existing bulk that has free storage capacity or it claims a new bulk on that node from the monitor. The maintenance unit then tracks the storage capacity available within the bulk and stores the newly put object to an available block. It then adds the ID-location tuple $< i, loc_i >$ to the Cuckoo table which is stored as part of the maintenance unit. Note that loc_i is a tuple including both IP and block addresses. It constructs the lookup unit of all $< i, loc_i >$ tuples and deploys the lookup unit to a node with sufficient DRAM resources. The lookup unit will be updated whenever additional objects are put into the system. When the number of objects within an active maintenance unit reaches a threshold, the maintenance unit is stored to the SSD and it becomes immutable (inactive). An inactive maintenance unit only needs to be accessed in the case of rare situations, such as large-scale object relocation. Performing put, get, modify, and delete operations no longer requires the access of an inactive maintenance unit. Hence only one active maintenance unit is running in the DRAM of each server at a time.

Both the monitor and maintenance units are small enough to be hosted on different storage nodes as long as the nodes have available DRAM space. Replicated monitors and maintenance units can also be deployed in this way to achieve fault tolerance. For this, replicated copies of each object are stored in multiple nodes. The ID-location mapping is then extended to $< i, loc_1, loc_2, loc_3 >$ for 3 copies in 3 different locations.

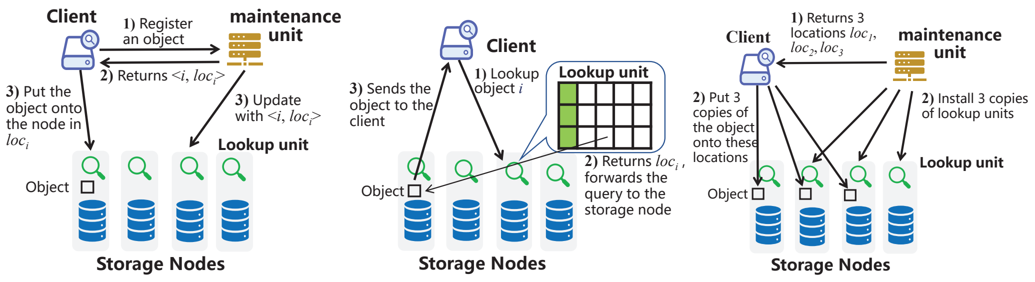
Lookup units. Lookup units respond to clients' object get and modify requests. Every lookup unit is resident in DRAM of a storage node. It returns the physical location $loc_i =< IP_i, block_i >$ of the requested object i and forwards the request to the corresponding node. For fault tolerance, replicated lookup units can run on multiple nodes.

Storage nodes. The storage space of a node is divided into blocks. When receiving a get request forwarded by one of the lookup units, the storage node returns the corresponding data to the client based on the block address $block_i$. When receiving an modify request, the storage node sends a message to the client directly, indicating the update was successful. Storage nodes also respond to put, relocate, or delete requests to objects. For fault-tolerance, replicated copies of an object can be stored in multiple nodes.

Clients. A client may or may not be in the same data center with the storage. For example, the clients of the object database of a social network are the web servers in the same cloud. Smash provides a client library for accessing the object storage system. Like the interfaces in existing key-value stores, the client can request the lookup units or maintenance units to put, get, relocate, or delete objects.

4.2 System Initialization

Objects and nodes can be incrementally deployed to a system running Smash. The number of inactive maintenance units and lookup units depends on the system size. The monitor and all maintenance units can be hosted by a server whose resources are not necessarily rich, possibly with one or two backup servers. The lookup units are hosted by the storage nodes with very little DRAM cost. When a storage node joins, the monitor notifies the active maintenance unit about the node's IP address and may assign the node's bulks to the active maintenance unit.


Client
1) Register an object
maintenance unit
2) Returns <i, loci>
3) Put the object onto the node in loci
3) Update with <i, loci>
Lookup unit
Object
Storage Nodes
Client
3) Sends the object to the client
1) Lookup object i
Lookup unit
Object
2) Returns loci, forwards the query to the storage node
Storage Nodes
Client
1) Returns 3 locations loc1, loc2, loc3
maintenance unit
2) Put 3 copies of the object onto these locations
2) Install 3 copies of lookup units
Lookup unit
Object
Storage Nodes

(a) Put an object. (b) Get an object. (c) Replicates for failure-tolerance.

Fig. 5. The object operations of Smash.

Smash sets the maximum number of objects per shard as α. Each maintenance unit then contains a Cuckoo hash table with $\lceil \frac{1}{0.95\times 4}\alpha \rceil = \lceil 0.263\alpha \rceil$ buckets with each bucket containing 4 slots. The reason behind this is that the total number of slots is then $\frac{1}{0.95}\alpha$, which can store all α ID-location pairs with the table's load factor up to 95%. According to theoretical studies [16, 19], insertions to a Cuckoo hash table of load factor up to 98.03% is asymptotically almost surely (a.a.s) successful. We use 95% to avoid hitting the tight threshold. The lookup unit has the same number of buckets.

Each client stores the mappings of every shard ID s_i to the IPs of all nodes that host the lookup unit of shard s_i, which takes a few MBs. Storing IPs of the nodes is also necessary for all existing object stores such as Ceph [1] and MapX [40].

4.3 Operations of Smash

Smash supports operations to put, get, modify, relocate, and delete objects. We describe the operations in the following. For ease of presentation and illustration, we first show the operations on one copy of each object. We then extend the operations with replicated copies for fault-tolerance.

Put. Figure 5a shows the steps of putting a new object i. The client registers the new object to the active maintenance unit (Step 1). The maintenance unit determines the location $loc_i =< IP_i, block_i >$ to store the object, based on the application requirements and node availability. It then tells the client the tuple $< i, loc_i >$ (Step 2). The maintenance unit updates the lookup unit and tells the update to the node that hosts the lookup unit (Step 3). This update takes $O(1)$ time and $O(1)$ communication bits in expectation [37]. At the same time, the client sends the object i to IP_i, the IP of the node to store i. The node IP_i will store i to block address $block_i$ (Step 3).

Get. Figure 5b shows the steps of getting an object i. The client finds the shard ID k from the ID i of the object it wants to get. Recall that each client maintains the mapping of every shard ID k to the IPs of nodes that host the lookup unit of shard k. It then sends a lookup request of i to the lookup unit (Step 1). The lookup unit returns loc_i by looking up the ID i. Then it forwards the lookup request to the node IP_i that stores i along with the block address $block_i$ included in loc_i (Step 2). The node IP_i gets the object from $block_i$ and sends it to the client (Step 3).

Delete. The process of deleting an object i can re-use the Steps 1) and 2) of Get. The difference is that in Step 3) the node that stores i just deletes i and sends a confirmation message to the client. Note this process does not need the involvement of the maintenance unit and the lookup unit does not need to change. In the lookup unit, if i has been removed and never be queried, the correctness of the lookup unit of MPH will not be affected. The maintenance unit needs to guarantee that an ID will not be assigned twice to different objects, which can be easily achieved.

Modify. The process of modifying an object i can re-use the Step 1) of Get. The difference is that in Step 2) when the lookup unit gets loc_i, it does not forward the request to the node IP_i but sends loc_i back to the client. Hence the client can directly contact the node IP_i to modify the content of i. This process does not need the involvement of the maintenance unit because the storage location of i does not change. Note that a "`Modify`" in `Smash` and an "update" in Ludo have completely different meanings. An "update" in Ludo means inserting, deleting, or changing a key-value pair. The correctness of insertion cannot be guaranteed without an active maintenance unit. A "`Modify`" in `Smash` is to change the content of an object. The key-value pair of this object from Ludo's perspective is the ID-location pair, which does not change for such an update. Without an active maintenance unit, an update of `Smash` is still always successful.

`Relocate`. A relocation happens rarely compared to the above operations. Each relocation is initiated by the monitor, rather than the clients. It happens when the monitor wants to further optimize the placements based on application requirements such as locality and load balance. Note that when each object is placed for the first time, its location is already optimized by the maintenance unit. Hence relocation may happen once during a long time period (such as several days). A relocation might change the maintenance and lookup units in multiple shards and hence some inactive maintenance units may need to be loaded to DRAM and updated at this point. Since there are fewer relocations happening during a long time period, the monitor can process relevant objects in one shard after another. Since the latency of relocation is not sensitive, changing inactive maintenance units will not introduce much DRAM cost. This is the only case where an inactive maintenance needs to change.

4.4 Replication for failure-tolerance

The above description of `Smash` operations assumes one copy of each lookup unit is running in the system and one copy of each object is stored. In practice, it is common to have replicated lookup units and replicated copies of objects to tolerate failures. In addition, the server that hosts the monitor and maintenance units also has two backup servers. The main server keeps synchronizing with the backup servers.

As shown in Figure 5c, we let `Smash` store 3 copies of each object and place 3 copies of a lookup unit to different nodes. When a new object is registered, the maintenance unit determines 3 storage locations, usually on nodes in different failure domains. `Smash` is compatible to arbitrary failure-tolerant placement strategies since it provides full flexibility. Hence the ID-location pair is extended to $< i, loc_1, loc_2, loc_3 >$. The lookup result by querying a lookup unit will be three locations instead of one. After knowing the three locations, the client will send three copies of the new objects to the three locations respectively. In addition, the same lookup units of a shard are hosted by three nodes and the location of an object can be obtained by querying any of the lookup units. At each client, the shard ID to the IP of the lookup unit is extended to three IP addresses of the three nodes hosting the lookup unit. For `Put`, `Delete`, and `Modify`, a client needs to communicate with all three nodes that store the object. For `Get`, a client can contact any of the 3 IPs that host the lookup unit, to get the object.

4.5 Failure handling

`Smash` includes recovery methods from node failures and server failures. We briefly describe them.

Node failure. A storage node continuously sends heartbeat messages to the monitor server during operations. When the server does not receive heartbeat messages from a node for a certain period or the maintenance unit or clients report the failure of reaching a node to the server, the server concludes that the node has failed. For a failed node, its DRAM is erased but the disk can return to operations after fixing the problem. Hence it is important to recover the lookup units running in the DRAM of the failed node. When an inactive maintenance unit is stored in an SSD, its immutable lookup unit is also persisted inducing only minimal storage cost. The monitor maintains the mappings of every lookup unit to its hosting nodes and finds out all lookup units that ran on the failed node. These lookup units can be loaded from the SSD. When the failed node returns back to work and the system is rebooted, the monitor then sends the lookup units to it. If the failed node cannot be fixed, the monitor will send the lookup units and the objects on the failed node to other nodes for hosting.

Server failure. The server running the monitor and maintenance units keeps sending heartbeat messages – including any synchronization – to the backup servers. When the server fails, it will be detected by the backup servers which will replace the failed instances. Clients detect the failure when TCP connections cannot be established with the previously known server, forcing them to choose a new replica.

4.6 Management Elasticity

So far we assume that each shard includes up to α objects and α is also the number of objects that a standard node can store. For example, when the storage capacity of a node is 16TB and each object is 100KB, $\alpha = 160$ million. Each shard has one maintenance unit, constructing one lookup unit that is responsible for the α objects. So in expectation, each node will host one lookup unit.

In practice, node capacities may be heterogeneous and α can be any value as the objects of the same shard can spread across different storage nodes. For a smaller α, each lookup unit costs smaller memory, hence we can allocate lookup units to nodes with available DRAM resources, with finer-grained management. However, smaller α also causes longer shard IDs and hence more bits in object IDs should be used for shard IDs, which indirectly increases the cost of maintenance units.

In addition, the maintenance unit of a shard can further make β sub-shards and construct a lookup unit for each sub-shard. Which sub-shard an object belongs to can be determined by hashing the object ID.

4.7 Analysis

We analyze the DRAM cost and time complexity of `Smash`.

DRAM cost. The DRAM cost of `Smash` has three types: 1) DRAM for the monitor. There is only one monitor needed in the whole system. 2) DRAM for the maintenance unit. There is only one active maintenance unit running in DRAM for the whole system and all other maintenance units are immutable and stored in SSD. 3) DRAM for lookup units. Each storage node needs to host one lookup unit or multiple ones for replicated lookup units.

1) Monitor. Assume the system has n storage nodes, each of which holds p blocks on average. There are k shards and each shard contains up to α objects. The length of a shard ID is $l_s \geq \lceil log_2(k) \rceil$ bits. Each object is stored in three different nodes. The monitor manages storage availability at the bulk level and each bulk contains 256 blocks. It also maintains the load percentage of each node, assuming each percentage number costs 10 bits. Thus the space required to keep the states in the monitor is $n \cdot \frac{\alpha}{256} \cdot l_s + 10n$ bits. Note that there is only one unique monitor in the whole system, running with a couple of replicas for failure tolerance.

2) Maintenance unit. Let l be the length of an object ID i, l_n be the length of a node ID, assumed to be the l_n-bit suffix of IP addresses, and the bucket address is $\lceil \log_2 p \rceil$-bit. The length of a location loc_i is $l_n + \lceil \log_2 p \rceil$ bits and the length of an ID-to-locations tuple is $l + 3(l_n + \lceil \log_2 p \rceil)$ bits, assuming 3 replicas. Each bucket costs $5 + 4l + 12(l_n + \lceil \log_2 p \rceil)$ bits. Hence the Cuckoo table of a maintenance unit costs $\lceil 0.263\alpha \rceil [5 + 4l + 12(l_n + \lceil \log_2 p \rceil)]$ bits, as it includes $\lceil 0.263\alpha \rceil$ buckets. The classifier costs 2.33α bits according to [37]. Therefore in total a maintenance unit costs $2.33\alpha + \lceil 0.263\alpha \rceil [5 + 4l + 12(l_n + \lceil \log_2 p \rceil)]$ bits.

3) Lookup unit. For each lookup unit, a bucket costs $5 + 12(l_n + \lceil \log_2 p \rceil)$ bits. Hence the lookup table costs $\lceil 0.263\alpha \rceil [5 + 12(l_n + \lceil \log_2 p \rceil)]$ bits. Together with the classifier, a lookup unit needs $2.33\alpha + \lceil 0.263\alpha \rceil [5 + 12(l_n + \lceil \log_2 p \rceil)]$ bits.

Suppose a large storage system includes $n = 10$ thousand nodes, each of which includes $p = 1$ million blocks, and $k = 10$ thousand shards, each of which includes $\alpha = 40$ million objects. An object ID has $l = 160$ bits. A shard ID has $l_s = 20$ bits. A node ID has $l_n = 20$ bits. In this setting, the number of shards is set to be equal to the number of nodes, so each node hosts one shard's lookup unit. The monitor costs 391MB, a maintenance unit costs 1.5GB, and a lookup unit costs 680MB. In this setting, a server with 4GB DRAM is sufficient to run the monitor and the maintenance units and each node only needs <1GB DRAM to run the lookup units. For a directory-based method, it needs a directory of 9TB DRAM on a metadata server for $10K \times 40M = 400B$ ID-to-node mappings. It also needs at least 900MB DRAM on each node for the local directory to support ID-to-block mappings.

Lookup and update complexity. The time complexity of querying a lookup unit is a small constant. Each lookup requires 3 memory accesses: two for the classifier and one for the bucket [37]. There is a small probability that the bucket seed overflows, hence the average number of memory accesses is 3.016.

The time complexity of adding one object to the maintenance and lookup units is also $O(1)$ in expectation. All three parts of an update operation, namely adding an ID-location mapping to the classifier, adding the mapping to the Cuckoo table, and updating the lookup unit, cost either $O(1)$ or amortized $O(1)$ time [37]. The communication cost to updating a lookup unit (in bits) is also $O(1)$ and in practice around 100 bits excluding the packet headers.

The time complexity of querying a lookup unit and adding one object are both $O(1)$ in expectation.

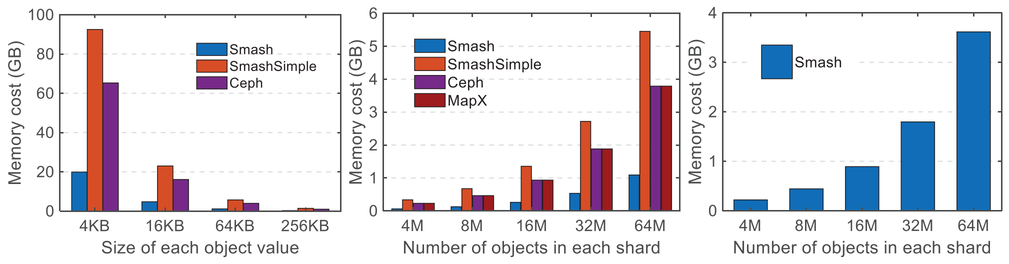
5 PERFORMANCE EVALUATION

In this section, we show the performance of `Smash` using an implemented prototype system and simulations. We compare `Smash` with `SmashSimple`, Ceph v12.2.0 [1] and a recent work MapX [40]. We also compare `Smash` with a directory-based version `SmashSimple`, a baseline solution that supports flexible placement. The difference between `SmashSimple` and `Smash` is that `SmashSimple` uses the Cuckoo hashing table as the main data structure used in lookup units. Full object IDs and data positions are stored as key-value pairs to support fully flexible placement. Ceph is a classic object storage system whose placement and lookup algorithm is CRUSH [43], which is a hybrid hashing-based scheme as described in Section 1. MapX is an extension of CRUSH, which maps storage nodes added to the system at different times into different layers. Getting/putting new objects into different layers with varied timestamps can reduce data movement or migration as the storage system expands. Note that both Ceph and MapX use Bluestore [2] (based on RocksDB [3]) to maintain local indices on each node. Each object's ID and block address are maintained in the LSM-tree-based database for metadata access.

5.1 Methodology

Hardware. The testbed consists of eight servers from a public cloud CloudLab [8]. Each server is equipped with two Intel E5-2630 v3 8-core CPUs at 2.40 GHz, 128GB ECC Memory, one Intel DC S3500 480 GB 6G SATA SSDs, and a dual-port Intel X520-DA2 10Gb NIC. These machines run Ubuntu 18.04 LTS with Linux kernel 4.15. In fact, `Smash` can run on much cheaper nodes with weaker resources.

Testbed configuration. We denote the eight servers as $S_0, S_1, .., S_6, S_7$. S_0 serves as the server to host the monitor and maintenance units, $S_1, .., S_6$ serve as storage nodes and S_7 serves as clients. `Smash` places the lookup units evenly on the 6 storage nodes, although in the design there is no limit to the number of lookup units running simultaneously on each storage node as long as its resource permits. To test Ceph and MapX, we use S_0 as the administrator and monitor of the system, which monitors the nodes' status. We use *ceph-deploy* to build the testing system first. For MapX, we separate the storage nodes $\{S_1, S_2, S_3\}$ and $\{S_4, S_5, S_6\}$ into two different layers. The number of placement groups is set to 128 as recommended in [40]. For Ceph and MapX, we use the C++ interfaces released in librados [1] to implement the operations, including putting, getting, updating,



(a) DRAM cost per node by varying the value size (b) DRAM cost per shard under the setting of Ceph (c) DRAM cost of maintenance unit

Fig. 6. DRAM cost comparison

and deleting objects. In addition, we set the number of copies of each object in each storage system to 3. We set that each object ID has 320 bits and a shard ID has 20 bits unless otherwise stated.

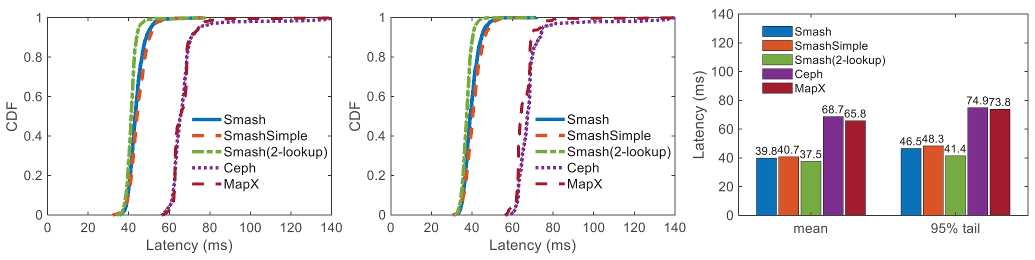
Workloads. We use both uniform and Zipfian distribution object query workloads and the Zipfian workload is modeled after real-world access patterns observed at Facebook [9]. The queried objects in uniform workload are generated uniformly randomly without any bias. Correspondingly, the Zipfian workload is generated with a biased parameter α (<1), containing a few popular objects. These kinds of workloads are used in recent works [28, 32] for benchmarking the key-value storage systems. In the evaluations, the client (S_7) will generate and store 10 thousand objects first and put them to the storage nodes. The contents of the objects are generated randomly. In the following evaluations, each operation (such as `Put` and `Get`) is conducted at least 1000 times with different objects in different locations. Each set of experiments is run for at least five times and the average performance is reported. In addition, half of the objects are with each of the two layers with different timestamps in MapX.

5.2 DRAM cost

Since we decouple `Smash`'s metadata layer into maintenance units and lookup units, they can run in different nodes with heterogeneous hardware. Memory-efficient lookup units could be put on any node independently, serving for object requests. For Ceph and MapX, we show the DRAM cost of the local lookup engine in Ceph/MapX with the assumption that storage nodes only have DRAM as the fast-accessing memory layer. We also compare `Smash` with a directory-based version `SmashSimple`. The difference between `SmashSimple` and `Smash` is that `SmashSimple` uses the Cuckoo hashing table as the main data structure used in lookup units. Full object IDs and data positions are stored as key-value pairs to support fully flexible placement.

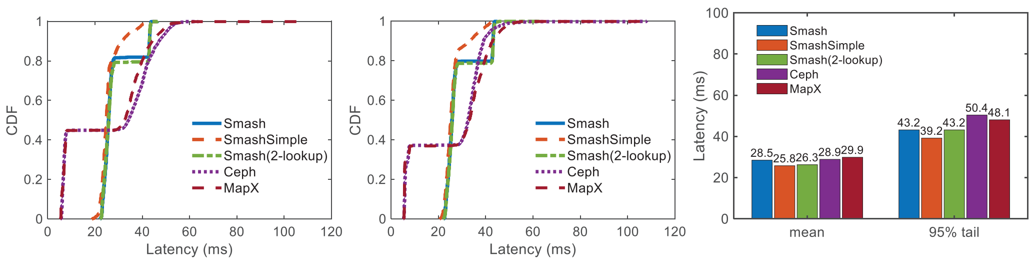
We first compare the DRAM cost per node by varying the average size of objects – and hence the number of objects per node. Each node has a storage capacity of 4TB and the key length is 40 Bytes according to the results in [9]. The object size varies from 16KB to 1MB, because in practical key-value applications such as those from Facebook and Twitter [9, 46], the sizes of most values are much smaller than 1MB, where around 30% of the values in the ETC workload of Facebook are smaller than 4KB [9]. `Smash` can even support smaller value sizes than 16KB but other methods could exhaust DRAM. Compared to `SmashSimple`, `Smash` can reduce the DRAM cost per node by 80%. Specifically, when the value size is 16KB, the DRAM cost is reduced from over 20GB to less than 5GB per node. When the value size is 64KB, the DRAM cost is reduced from 4.8GB to 1.1GB per node. The DRAM cost of Ceph is also significantly higher than that of `Smash`.

As shown in Fig. 6b, we vary the number of objects in one shard from 4M to 32M to show the DRAM cost per shard. In our default setting, the number of shards is equal to the number of nodes. Hence the DRAM cost per shard is also equal to the DRAM cost per node in this setting. We find that



(a) `Put` latency w/ uniform workload. (b) `Put` latency w/ zipfian workload. (c) Average latency and 95% tail latency.

Fig. 7. Latency for putting objects.



(a) `Get` latency w/ uniform workload. (b) `Get` latency w/ zipfian workload. (c) Average latency and 95% tail latency.

Fig. 8. Latency for getting objects.

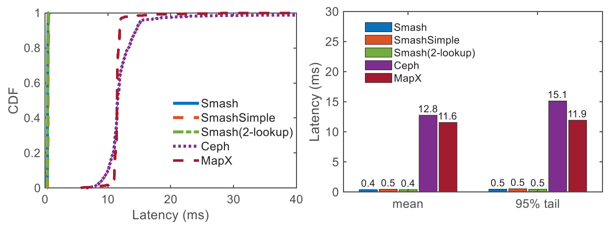
`Smash` can always achieve over 70% lower DRAM cost compared to others. Hence `Smash` achieves a smaller DRAM cost than existing methods.

Fig. 6c shows the DRAM cost of the active maintenance unit by varying the number of objects in each shard, α. Even with 32M objects per shard, the DRAM cost is no more than 1.8GB. The monitor needs an extra 100 to 400MB. Hence the server of `Smash` does not to be very powerful.

5.3 Latency of storage operations

In this subsection, we show the testbed evaluation results of storage operation latency, including `Put`, `Get`, `Modify`, and `Delete`, under different distribution workloads. We first put 10 thousand objects in the cluster, and then the client issues 4000 operations for putting/getting/modifying/deleting objects with a replication factor of 3. In addition, to demonstrate that multiple lookup units can run simultaneously on different machines together in `Smash`, we show the performance of `Smash` with running one and two lookup units respectively.

`Put`. Fig. 7 shows the cumulative distribution (CDF) of `Put` latency for these three methods under different workloads. Fig. 7a shows their latency under the uniform workload. Fig. 7b shows their latency under the Zipfian workload. Fig. 7c shows their average and 95% tail latency under the Zipfian workload. We find that `Smash`'s latency is lower than that of Ceph and MapX. Note that these methods include different optimization stages and implementation details. For example, Ceph needs to perform cluster health status check. Both Ceph and MapX map objects to PGs first and then to nodes, which is a more time-consuming process. In `Smash`, the monitor maintains a heap for the load status of each disk, and then the storage objects and replicas can be placed directly into the lowest-loaded nodes. Hence it is unclear that `Smash`'s latency is definitely shorter than that of Ceph and MapX, but it is safe to conclude that all of them have short and similar latency in operations. `SmashSimple` also achieves similar results as `Smash` with a Cuckoo hashing table.



(a) Deleting latency w/ uniform workload.
(b) Average latency and 95% tail latency.

Fig. 9. Latency for deleting objects.

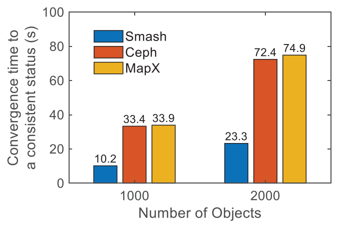
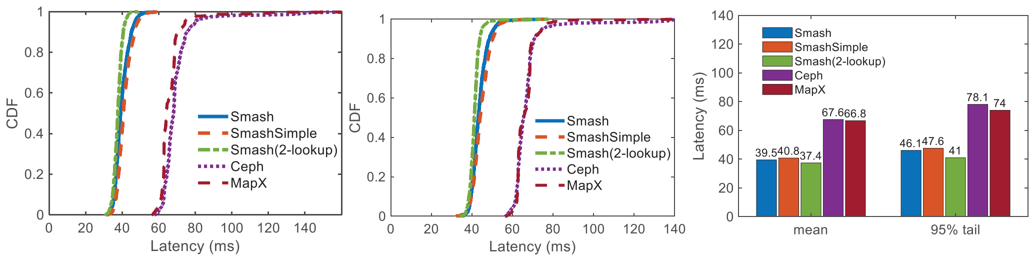


Fig. 10. Convergence time for one node failure/removal.



(a) Modify latency w/ uniform workload.
(b) Modify latency w/ zipfian workload.
(c) Average latency and 95% tail latency.

Fig. 11. Latency for modifying objects.

Smash with two lookup units performs better than that with a single lookup unit, however, their performance is similar.

Get. Fig. 8 shows the CDF of Get latency under the uniform (Fig. 8a) and Zipfian (Fig. 8b) workloads and showing the average and 95% tail latency (Fig. 8c). Again, Smash achieves the smallest average and tail latency compared to Ceph and MapX, although all of them are quite fast. Ceph and MapX have similar Get latency. Their latency below the 30th percentile is smaller than that of Smash. Note that in practice the latency of all methods could be shorter due to DRAM caching but we do not enable caching in this set of experiments. However, Ceph and MapX have relatively long tail latency. The main reason is the iterations of calling the 'select' function of CRUSH. In Smash, the trailing delay shown in the picture is mainly caused by the fallback table of the lookup unit structure. The results for SmashSimple are similar to those of Smash, with SmashSimple slightly faster than Smash in high-percentile results.

Modify. Fig. 11 shows the Modify latency of Smash and SmashSimple compared to Ceph and MapX. The results are very similar to those of Put in Fig. 7, because both operations need to write new object contents to the storage. The latency of both Smash and SmashSimple are very close and with similar values of Ceph and MapX.

Delete. Fig. 9 shows the CDF of Delete latency under the uniform workload – the Zipfian workload makes almost no difference hence the results are not shown here. Smash performs an object deletion in less than 1 ms on average, whereas Ceph and MapX have latency greater than 10 ms. An object deletion of Smash is effective as soon as the maintenance unit receives the request and stores the request. The removal of the corresponding entry from the lookup unit can be done immediately. The object deletion from the stored disk can be delayed. The system may even mark the block as empty and a future object could overwrite the block. On the other hand, Ceph and MapX need to spend time on cluster health checking, which is a time-consuming process.

# Objects	Smash	MapX	Ceph
1000	0.02s	22.3s	69.3s
2000	0.03s	53.0s	115.4s

Table 4. Convergence time for adding one node.

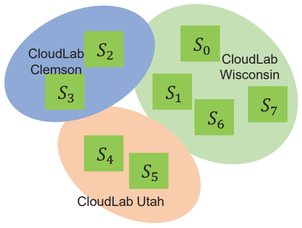
5.4 Recovery after node addition and failure

One important problem of classic object storage is when a node addition or removal happens, objects need to be relocated or replicated to make the system converge to a consistent state that satisfies hashing results. We evaluated the convergence time of the testbed under node additions and removals. In the beginning, we put all objects onto the storage nodes and then reduce or increase the number of nodes. Then the cluster needs to relocate or replicated affected objects. We compare the convergence time of the three methods, by varying the number of objects.

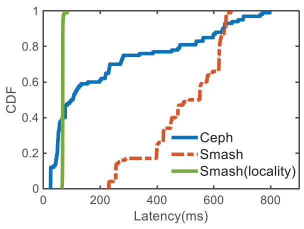
Node failure. The system contains six nodes that store objects. We make a node fail from the system. The monitor server detects this event and triggers the recovery process. It assigns the objects of the failed node to other nodes and asks the nodes with another copy of those objects to send a copy of each object to the new location. In the end, every object still has three copies stored in the system. We record the time for this process to finish. For Ceph (same with MapX), if we just use the released API to handle a node failure, Ceph will make extra optimizations (e.g., "health check") while storing more copies of the objects. To make a fair comparison with Smash, we dump the IDs of all objects in the failed node. Then the server uses this list to put the objects to other nodes based on the CRUSH algorithm. We record the time taken to complete this task.

Figure 10 shows the convergence time of the three methods to reach a consistent state under a different number of distinct objects stored in the cluster. When there are 1000 distinct objects and each object has three copies, every node stores around 500 objects. From the results, we find Smash takes only less than one-third time to converge, compared with Ceph and MapX. For all methods, processing each object on the failed node consists of two parts: 1) finding a new location for every object and 2) then putting a new copy for every object. Part 2 takes the same time for all methods. Hence the difference was due to Part 1. Due to the full flexibility of placement, Smash can easily decide the new locations of these objects. However, Ceph needs to map the objects to PGs and then to physical nodes. If a conflict occurs (e.g., a copy of the object is already stored in the target node), the "select" function needs to be run iteratively. When the number of distinct objects in the cluster is 2000, we can see that the difference in convergence time between Smash and Ceph/MapX increases.

Node addition. We initialize the system with five storage nodes and then add one node to the system. Smash does not need to move stored objects to the new node unless the existing nodes are overloaded. Ceph and MapX require data migrations to maintain load balance and meet the new hashing-based mapping rules. From the paper of MapX [40] we know that in a large cluster, adding one node will affect about 10% of the existing PGs, and the new PG-node mapping needs to be re-established. The time required to converge to a consistent state of the three methods can be seen from the table 4. Smash takes the shortest time because it does not need to relocate stored objects. The time required for Smash is mainly composed of 1) the storage node registers to the monitor and 2) the monitor changes its current state and notifies other units in the system, which takes less than a second. MapX needs to place the added node under a new layer, which affects about half of PGs. The number of objects moved in MapX is much smaller than that in Ceph, thus the convergence time of MapX is shorter than that of Ceph, but still orders of magnitudes higher than that of Smash. Note that although Smash does not relocate the stored objects when a new node joins, Smash can still achieve load balance by setting the new node with a high weight in future node selection for new objects.



(a) Cloudlab setting for locality experiments.



(b) CDF of Get latency.

Fig. 12. Cross-cluster experiments on latency.

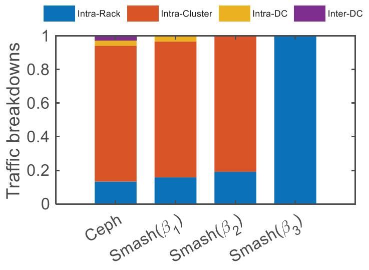


Fig. 13. Traffic types breakdown in Ceph and Smash in the number of flows.

5.5 Benefit from flexible placement: reduced traffic

We demonstrate one of the benefits of Smash's flexible placement, in particular, from object locality. We run two sets of experiments. In the first set, we set the optimization goal as minimizing Put and Get latency. In the second set, we set the optimization goal as minimizing long-distance network traffic, such as inter-rack and inter-datacenter traffic. It is a well-known design goal of data and virtual machine placement in datacenters [10], because long-distance network traffic could cause congestion at upper-level switches or routers and thus affect the performance of the entire network.

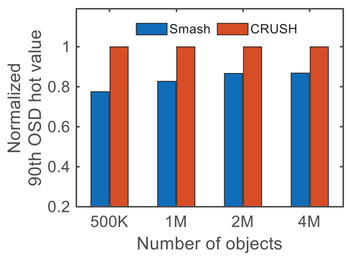
We build the testbed with eight machines in three different clusters of CloudLab running in Wisconsin, Utah, and South Carolina, respectively, as shown in Figure 12a. We set a workload such that each client has a particular group of objects to put and get. Smash can place objects close to their clients. As shown in Figure 12b, both Ceph and Smash have high latency due to their unawareness of object locality and high inter-cluster traffic. Smash can put objects arbitrarily according to the application requirements. Hence its latency is much shorter.

We run a set of simulations to show the advantages of Smash in minimizing long-distance network traffic. We use the traffic workload using the traffic generator by Mellanox [4], which is based on the traffic characteristics from Facebook [35]. We simulate a network topology of n_d datacenters; each datacenter includes n_c clusters, each cluster includes n_r racks, and each rack includes n_h nodes. We classify four types of traffic in a generated workload: 1) intra-rack (with α_r flows), the source and destination are in the same rack; 2) intra-cluster (with α_c flows), the source and destination are in the same cluster but not the same rack; 3) intra-DC (with α_d flows), the source and destination are in the same datacenter but not the same cluster; and 4) inter-DC (with α_c flows), the source and destination are not in the same datacenter. Each flow is to get one object. Also the capacity for a particular host, the number of objects it can store is β of the number of all objects. We test three values of β: $\beta_1 = \frac{n_h(1-\alpha_r)}{\alpha_c+\alpha_d+\alpha_c}$, $\beta_2 = \frac{n_h(1-\alpha_r)}{\alpha_c+\alpha_d}$, $\beta_3 = \frac{n_h(1-\alpha_r)}{\alpha_c}$. We use a greedy bin-packing algorithm to place the object close to the client that wants to retrieve it. Obviously, inter-DC is the most expensive and non-preferred type of traffic, followed by intra-DC, and intra-cluster. Intra-rack traffic is the most preferred.

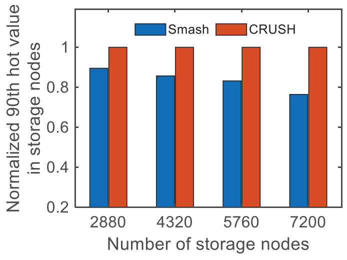
The results are shown in Figure 13; we find Smash can significantly reduce the amount of inter-DC traffic compared to Ceph. Ceph cannot optimize long-distance traffic. Depending on the values of β, Smash can also reduce intra-DC and intra-cluster traffic. When we use β_3, all flows are intra-track.

5.6 Benefit from flexible placement: load balancing

We evaluate Smash and CRUSH on loading balancing for a cluster with 7200 storage nodes. We generate obj_num objects with different hot values (the hotness ranges in $[0, 100]$), and assign them with hashing algorithms shown by CRUSH in Ceph. Smash can move objects to specific storage nodes based on the hot values with the greedy algorithm to make all nodes have similar and balanced sums of hot values with small variances.

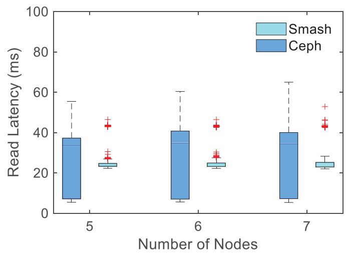


(a) Loading balancing with a different number of objects per node.

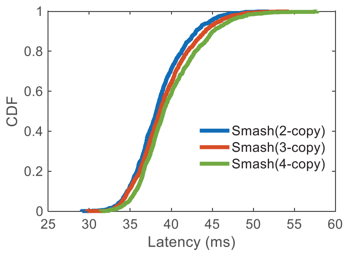


(b) Loading balancing with different number of storage nodes.

Fig. 14. Evaluation of load balancing.



(a) `Get` latency vs. the number of nodes.



(b) `Modify` latency with different numbers of copies.

Fig. 15. Latency of Smash and Ceph

We define the load balancing metric as the normalized 90th percentile value among the sums of hot values of the storage nodes in the cluster. We further normalized the hot values by setting the 90th sum of hot values in CRUSH as 1. Fig. 14a shows the load balancing metric by changing the number of objects per node from 500K to 4M. The 90th sum of hot values in Smash is always lower than that of CRUSH, varying from 77% to 86%, showing better load balancing. In addition, we vary the number of storage nodes from 2880 to 7200 and show the comparison in Fig. 14b. When the number of storage nodes increases, the difference between the load balancing metrics also increases, indicating better load balancing for Smash. Note that Smash is able to support any algorithm for data migration and load balancing and here we only apply a simple greedy algorithm.

5.7 Varying system settings

Vary the number of nodes. We vary the number of nodes in our testbed from 5 to 7. Figure 15a shows the latency for `Get` operations of Ceph and Smash with different numbers of nodes. In this set of experiments, we do not apply any optimization of object locality. Smash always achieves low and stable latency, because the speed of the algorithm running for the lookup units is not affected by system size. Ceph's average latency is also stable, but the latency has a significant variance.

Vary the number of object copies. We vary the number of copies of the same object in the testbed from 2 to 4. A larger number of replicas usually bring more complexity to lookup unit synchronization under object modification. We evaluated the relationship between the number of copies and the latency of the `Modify` operation – the other operations are not impacted by the number of replicas. We can see from Figure 15b that when we increase the number of replica copies stored in Smash, the `Modify` latency increases correspondingly but very slightly. However, Smash is very quick to process `Modify` operations even with four copies, indicating its fast synchronization among lookup units.

6 RELATED WORK

Object storage [6, 42] is a type of storage systems that manage data as objects, where files are converted into one or more objects and stored on distributed storage nodes. This work discusses a placement and lookup method of general distributed storage but uses object storage as a study case.

Many file systems use *central directories* to store data-to-location mappings [14, 20, 25, 30, 34], where the ‘location’ can be the network address of a storage node. A directory needs to be run in the DRAM of one or more servers to support instant queries and these servers are called metadata servers in many file systems. Some object storage systems also apply this approach. Amazon S3 [33] allows each user puts its objects to a bucket and maintains the full object-to-bucket mappings. Lustre [15] and DAOS [29] are both object storage replying on meta servers to tell object locations. For large-scale object storage, the resource overhead for the directory is huge and hard to be replicated to avoid becoming a single point of failure. There are two main reasons: 1) the number of objects is big and 2) the size of each key is also long, even on the same scale of the object [9, 46]. Hence storing key-location mappings in the directory would cost a massive amount of DRAM space (e.g., > 400GB for 10 billion keys).

To avoid the scalability bottleneck on the central directory, many object stores use *hashing* to determine the object locations. Hence clients get object locations by calculation instead of lookups. OpenStack Swift uses consistent hashing [22] to determine the object locations. SoMeta [39] uses a Distributed Hash Table (DHT) [38] to manage metadata objects across multiple servers. Hashing determines object locations based on a pseudo-random and deterministic function rather than application requirements.Hence hashing fails to meet these application requirements. CRUSH [43] is the hashing-based placement and lookup method used in Ceph [1, 36, 42], an open-source object storage system. CRUSH aims to mitigate the problems caused by simple hashing, including load imbalance, managing failure domains, and high data migration cost in response to the addition and removal of nodes, using a ‘cluster map’. However, it still cannot completely solve these problems and does not meet other application requirements such as data locality. *Hybrid approaches* [40, 41] add some flexibility of object locations based on practical concerns such as load balancing, but they still cannot achieve full control of object locations. Storage systems exploit flexible data placement on other layers of the storage stack. For instance, SSDs utilize the flash-translation layer (FTL) as a level of indirection to enable load-balancing [27] and to increase the lifetime of storage systems [18, 21, 31]. These techniques are orthogonal to `Smash` as they address such problems only on the device layer but not on the storage-cluster level.

7 CONCLUSION

This paper presents Smash, a novel placement and lookup method for large-scale storage systems. Compared to existing object storage such as Ceph (CRUSH) and MapX which uses hash values to place objects, the key advantage of Smash is to achieve fully flexible placement, which allows the system to optimize the object locations based on application requirements. `Smash` needs very little DRAM resource and the per-node DRAM cost is lower than that of CRUSH and MapX. We implement `Smash` using a testbed running in a public cloud and demonstrate its advantages by comparing it with existing work. Our future work will focus on applying `Smash` in other scenarios such as edge computing.

ACKNOWLEDGMENTS

We thank our shepherd Timothy Zhu and the anonymous reviewers for their suggestions and comments. Y. Liu, S. Shi, and C. Qian were partially supported by National Science Foundation Grants 1750704, 1932447, and 2114113. M. Xie and H. Litz were partially supported by National Science Foundation Grants 1942754 and 1841545. M. Xie was supported by the Center of Research in Storage Systems at UC Santa Cruz.

REFERENCES

[1] https://docs.ceph.com/.

[2] https://docs.ceph.com/en/latest/rados/configuration/storage-devices/.

[3] https://github.com/facebook/rocksdb.

[4] https://github.com/mellanox/dctrafficgen.

[5] https://github.com/mongodb/mongo.

[6] https://github.com/openstack/swift.

[7] https://github.com/yliu634/smash.

[8] https://www.cloudlab.us/.

[9] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-Scale Key-Value Store. In *In Proc. of ACM SIGMETRICS* (2012).

[10] Ballani, H., Costa, P., Karagiannis, T., and Rowstron., A. Towards Predictable Datacenter Networks. In *In Proc. of ACM SIGCOMM* (2011).

[11] Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., and García-López, P. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (2019), pp. 41–54.

[12] Bel, O., Chang, K., Tallent, N., Duellmann, D., Miller, E. L., Nawab, F., and Long, D. D. E. Geomancy: Automated Performance Enhancement Through Data Layout Optimization. In *Proceedings of 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2020).

[13] Belazzougui, D., and Botelho, F. C. Hash, displace, and compress. In *Proc. of Algorithms-ESA* (2009).

[14] Borthakur, D., et al. HDFS Architecture Guide. *Hadoop apache project 53*, 1-13 (2008), 2.

[15] Braam, P. The Lustre Storage Architecture. *arXiv preprint arXiv:1903.01955* (2019).

[16] Cain, J. A., Sanders, P., and Wormald, N. The Random Graph Threshold for k-orientiability and a Fast Algorithm for Optimal Multiple-Choice Allocation. In *Proc. of ACM-SIAM SODA* (2007).

[17] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020), pp. 209–223.

[18] Chakraborttii, C., and Litz, H. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (2021), pp. 1–12.

[19] Fernholz, D., and Ramachandran, V. The k-orientability Thresholds for $G_{n,p}$. In *Proc. of ACM/SIAM SODA* (2007).

[20] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), pp. 29–43.

[21] Kargar, S., Litz, H., and Nawab, F. Predict and write: Using k-means clustering to extend the lifetime of nvm storage. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), IEEE, pp. 768–779.

[22] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *In Proc. of ACM SOTC* (1997).

[23] Klimovic, A., Litz, H., and Kozyrakis, C. Reflex: Remote flash = local flash. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 345–359.

[24] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 427–444.

[25] Li, S., Lu, Y., Shu, J., Hu, Y., and Li, T. LocoFS: A Loosely-coupled Metadata Service for Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), pp. 1–12.

[26] Li, X., Andersen, D., Kaminsky, M., and Freedman, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys* (2014).

[27] Litz, H., Gonzalez, J., Klimovic, A., and Kozyrakis, C. Rail: Predictable, low tail latency for nvme flash. In *Transactions on Storage (ToS)* (2021).

[28] Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., Jin, X., and Stoica, I. Distcache: Provable Load Balancing for Large-scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 143–157.

[29] Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., and Barton, E. DAOS and Friends: A Proposal for an Exascale Storage System. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 585–596.

[30] Lv, W., Lu, Y., Zhang, Y., Duan, P., and Shu, J. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 313–328.

[31] Purandare, D., Wilcox, P., Litz, H., and Finkelstein, S. Append is near: Log-based data management on zns ssds. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22).* (2022).

[32] Rashmi, K., Chowdhury, M., Kosaian, J., Stoica, I., and Ramchandran, K. EC-Cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 401–417.

[33] Amazon Web Services. Amazon simple storage service. https://aws.amazon.com/s3/.

[34] Ren, K., Zheng, Q., Patil, S., and Gibson, G. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE, pp. 237–248.

[35] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 123–137.

[36] Sevilla, M. A., Watkins, N., Maltzahn, C., Nassi, I., Brandt, S. A., Weil, S. A., Farnum, G., and Fineberg, S. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), IEEE, pp. 1–12.

[37] Shi, S., and Qian, C. Ludo hashing: Compact, Fast, and Dynamic Key-Value Lookups for Practical Network Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 4*, 2 (2020), 1–32.

[38] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[39] Tang, H., Byna, S., Dong, B., Liu, J., and Koziol, Q. Someta: Scalable Object-centric Metadata Management for High Performance Computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (2017), IEEE, pp. 359–369.

[40] Wang, L., Zhang, Y., Xu, J., and Xue, G. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020), pp. 1–11.

[41] Wang, Y., Li, C., Shao, X., Chen, Y., Yan, F., and Xu, Y. Lunule: An Agile and Judicious Metadata Load Balancer for CephFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–16.

[42] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[43] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. Crush: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), IEEE, pp. 31–31.

[44] won You, G., won Hwang, S., and Jain, N. Scalable Load Balancing in Cluster Storage Systems. In *In Proc. of ACM/IFIP/USENIX Middleware* (2011).

[45] Xie, M., and Qian, C. Reflex4arm: Supporting 100gbe flash storage disaggregation on arm soc. In *OCP Future Technology Symposium* (2020).

[46] Yang, J., Yue, Y., and Rashmi, K. V. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage* (2021).

[47] Yu, Y., Belazzougui, D., Qian, C., and Zhang, Q. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* (2018).

Received October 2022; revised December 2022; accepted April 2023