DOI:10.1145/3747582

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3747583

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

By Yi Liu, Shouqian Shi, Minghao Xie, Heiner Litz, and Chen Qian

ABSTRACT

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 uses 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 (less than 60%) compared to existing hash-based solutions, such as Ceph and MapX.

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,[1,23] social networks,[6] data analytics,[12] and serverless computing.[3] In such a system, each data file consists of one or more named objects 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.

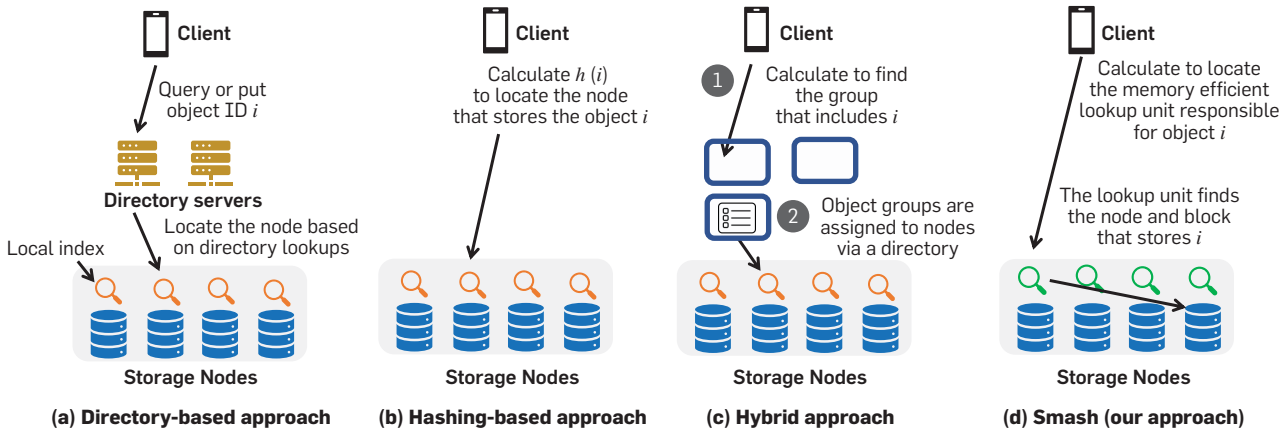
Object placement and lookup represent fundamental tasks that must be provided by any storage system. To manage objects on a massive scale, there are two typical object-placement and lookup strategies:

1. *Naïve directory-based approaches*, shown in Figure. 1a, 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 more than 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[7] and 40 bytes in Twitter[24] or Facebook.[2] For many applications, the data/ID space ratio is smaller than 10 for most data[24] and, as a result, many object storage systems choose an alternative option to implement object lookup.

2. *Hashing-based approaches*, shown in Figure 1b, place data to storage nodes based on the hash value of the ID $h(ID)$.[11,18,21] 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 original version of this paper, "Smash: Flexible, Fast, and Resource-efficient Placement and Lookup of Distributed Storage,"[15] was published in *POMACS* 2023.

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



Problems include placing replicas into the same failure domain, introducing load imbalance, and forcing data relocation when nodes join or leave the system.[21]

A common technique to address the above problems is to use hybrid approaches, as shown in Figure 1c, such as Ceph,[7,21] 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 Figure 1c, all objects are mapped to groups using hash computations, and then all groups are assigned to different nodes via directories. 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 Figure 1d, 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 resources 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 scales to 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 called Ludo hashing,[17] whose theoretical basis is dynamic minimal perfect hashing (MPH). 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 to 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 being inserted into the storage system, whereas deactivated maintenance units governing already placed objects only need to be updated when a large number 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[21]), the placement algorithm (Ceph[7]), and very recent work (MapX[19]), under different workloads. We show that Smash can achieve full placement flexibility, reduce the DRAM cost per node by greater than 60% compared to other solutions, and achieve low latency in putting/getting objects.

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 six million objects per node—greater than 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 datacenter.

2. OBJECTIVES AND ALGORITHM FOUNDATIONS

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

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 the header and objects can grow toward the middle. Clients are authorized users to access the objects, which may or may not be in the same cluster of the storage system.

The design objectives of Smash include:

- *Full placement flexibility*. Smash must support the placement of objects to arbitrary nodes, based on the application requirements for implementing fault tolerance, load bal-

ancing, data locality, and exploiting parallelism.

▸ *Low DRAM cost*. Smash needs to minimize metadata storage overheads and DRAM footprint to provide a low total cost of ownership (TCO).

▸ *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.

▸ *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 the data and the priorities of placement policies. We just name a few here:

▸ *Failure tolerance*. Some applications may require the replicas of some data to be in different failure domains to improve system robustness.

▸ *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.

▸ *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.[21] This requirement is particularly crucial for nodes with heterogeneous capacities and speeds because slow devices will become the bottleneck of overall performance.

▸ *Data placement*. Some workloads require special placement of data to optimize performance, such as those of high-performance computing.[4] 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 1 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 with the reported results in Wang et al.[19] Smash can take any object-to-node placement as the input and does not need data migration under node dynamics.

Table 1. Bandwidth cost for adding one node in CRUSH.

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

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, for example, greater than 80%.[17] The reason is the sizes of IDs are usually much longer than those of locations—Ceph[20] uses 16-byte IDs, and Twitter's average key length is around 40 bytes.[24] It could cost hundreds of gigabytes of DRAM for 10 billion ID-location mappings, and the majority proportion is used to store the IDs. For the methods that use 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 megabytes) needs to be run in DRAM. Hence, the DRAM cost of Smash is scalable to support an extremely large number of storage objects.

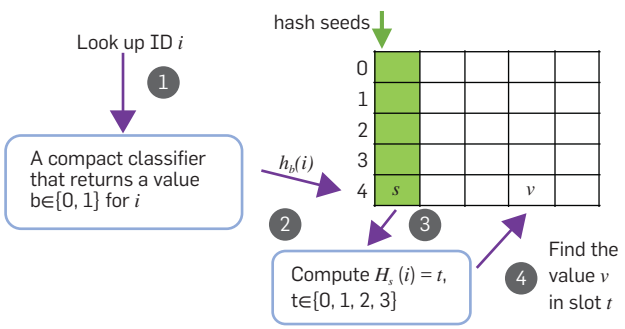
2.2. Ludo hashing.

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 use Ludo hashing[17] 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.[14]

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 Figure 2, the lookup structure consists of two stages. 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 a Bloomier filter.[25] 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 be-

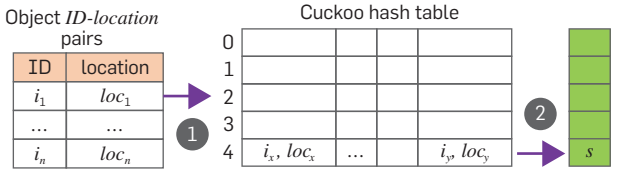
Figure 2. The lookup structure of Ludo hashing.



cause we have two independent hashing functions. The result $h_b(i)$ maps to a bucket (row) of a table shown on the right of Figure 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, $\mathscr{H}_s(i)$, 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 $O(1)$.

The maintenance structure. The maintenance structure is used to construct the lookup structure. As shown in Figure 3, it uses a (4,2)-Cuckoo hash table[14] 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 $\mathscr{H}_s(i)$ for the four keys in the bucket are different. Each seed is 5 bits long, and a very small portion of seeds longer than 5 bits 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 there is no collision for IDs 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*.

Figure 3. The maintenance structure of Ludo hashing.



3. APPLYING LUDO TO OBJECT STORAGE

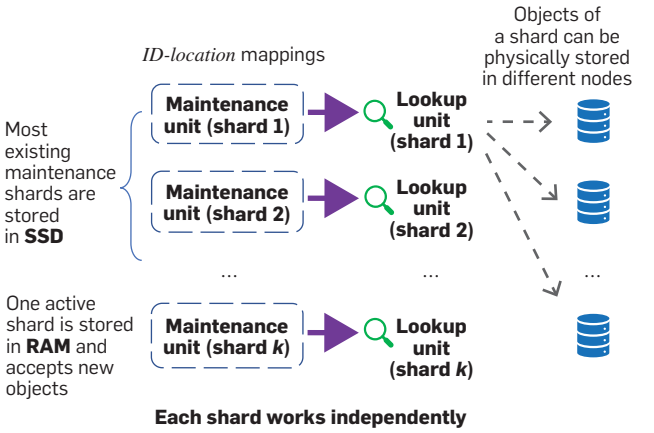
Using Ludo directly as a solution for the central directory could be an improvement over existing directories, but it still suffers from some crucial problems: The lookup structure still could be a bandwidth bottleneck, and the maintenance structure still introduces high DRAM space. Hence, we propose 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 Figure 4, all objects are divided into multiple shards. Each shard includes the objects put into the system within a period of time.

Figure 4. Shards of Smash.



In Smash, each maintenance structure is responsible for one shard that includes a fixed number (for example, 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,[7] 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 a design supports a wide range of applications as long as the stored data of these applications increases gradually over time without extensive deletion operations (for example, 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: Web applications that store user activity data, Internet of Things (IoT) applications that store the sensing and monitoring results from IoT devices, and log data of large systems. In fact, embedding an object's creation timestamp is very common in databases. For example in MongoDB,[16] each object ID is a 12-byte value that includes

a 4-byte timestamp representing 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 (for example, less than 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 (less than 250MB) even for eight 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 less then 60% DRAM per node compared to the local indices of CRUSH,[7] 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 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 update 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 resources: It requires one server to run the monitor and active maintenance unit and store the other maintenance units. The lookup units run in the DRAM of the storage nodes.

Suppose a large storage system includes $n = 10$ thousand nodes, $k = 10$ thousand shards, and 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 less than 1GB DRAM to run the lookup units.

Monitor. The monitor provides the following functions. 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 stores one or more objects. These sizes may vary for different applications. Bulk-level management enables the monitor to track whether each bulk has been assigned to an existing maintenance unit, though it does not track whether a block has been used or not. Each maintenance unit performs block-level management, 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.

The monitor also tracks the resource load of every node, including disk space, network bandwidth, DRAM, and CPU. The granularity of these loads is user-specified.

Lastly, 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 Won You et al.,[22] 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 in-

cluding 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 three copies in three different locations.

Lookup units. Lookup units respond to clients' object `get` and `modify` requests. Every lookup unit is resident in the DRAM of a storage node. It returns the physical location $loc_i = < IP_i, bloc k_i >$ 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 $bloc k_i$. When receiving a `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 datacenter 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 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.

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 four 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,[5,9] insertions to a Cuckoo hash table of load factor up to 98.03% are 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.

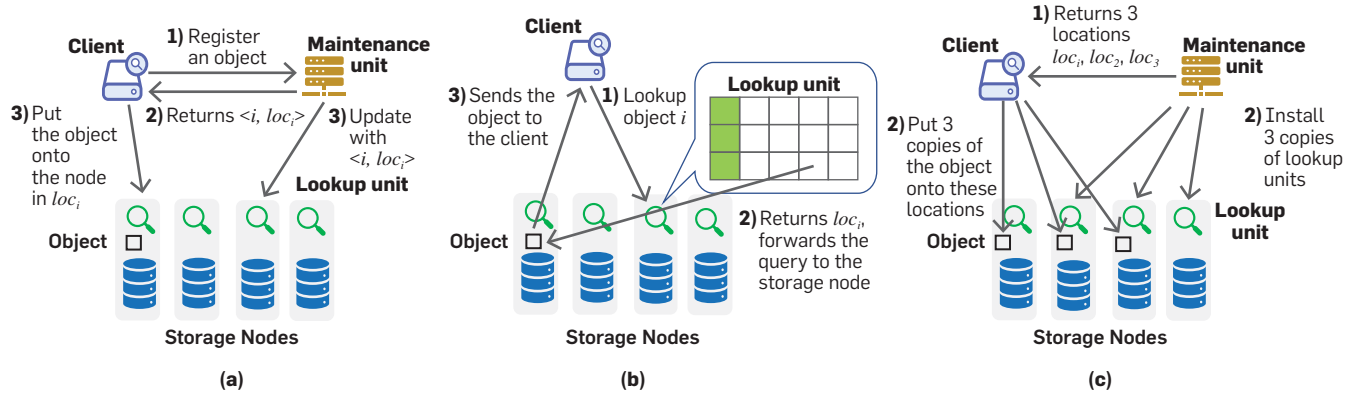
Each client stores the mappings of every shard ID s_i to the IPs of all nodes that host the lookup unit of shard s_i, which takes a few MBs. Storing IPs of the nodes is also necessary for all existing object stores, such as Ceph[7] and MapX.[19]

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 $loc_i = < IP_i, bloc k_i >$ 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.[17] 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 $bloc k_i$ (Step 3).

Figure 5. The object operations of Smash.



`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 $bloc k_i$ included in loc_i (Step 2). The node IP_i gets the object from $bloc k_i$ and sends it to the client (Step 3).

`Delete`. The process of deleting an object i can reuse 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 been 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 reuse 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 changes 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 unit needs to change.

4.4. Management elasticity.

So far, we assume 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, α = 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 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.

5. PERFORMANCE EVALUATION

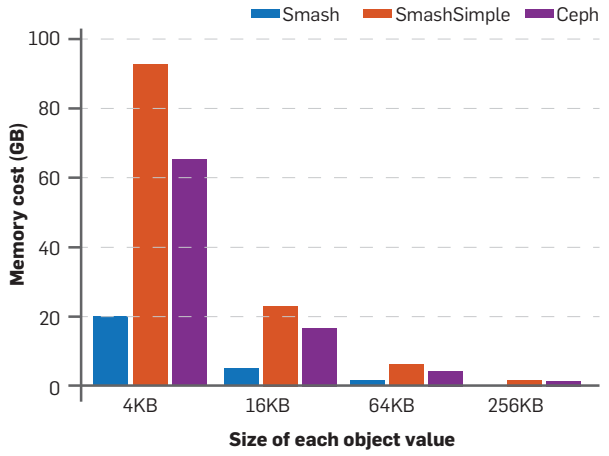
In this section, we show the performance of Smash using an implemented prototype system and simulations. We compare Smash with SmashSimple, Ceph v12.2.0,[7] and a recent work, MapX.[19] We also compare Smash with a directory-based version, SmashSimple, a baseline solution that supports flexible placement. The difference between SmashSimple and Smash is 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,[21] 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.

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 2.40GHz, 128GB ECC Memory, one Intel DC S3500 480GB 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 six 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 resources permit. 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 Wang et al.[19] For Ceph and MapX, we use the C++ interfaces released in librados[7] to implement the operations, including putting, getting, updating, and

Figure 6. DRAM cost per node by varying the value size.



deleting objects. We also 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.[2] 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. In the evaluations, the client (S_7) will generate and store 10,000 objects first and put them to the storage nodes. In the following evaluations, each operation (such as `Put` and `Get`) is conducted at least 1,000 times with different objects in different locations. In addition, half of the objects are with each of the two layers with different timestamps in MapX.

5.2. DRAM cost.

For Ceph and MapX, we show the DRAM cost of the local lookup engine in Ceph/MapX, assuming that storage nodes only have DRAM as the fast-accessing memory layer. We also compare Smash with SmashSimple. We first compare the DRAM cost per node by varying the average size of objects; 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 Atikoglu et al.[2] Object size varies from 16KB to 1MB, because in practical key-value applications, such as those from Facebook and Twitter,[2,24] 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.[2] Smash can even support smaller value sizes than 16KB, but other methods could exhaust DRAM. As shown in Figure 6, compared to Smash-Simple, Smash can reduce DRAM cost per node by 80%. Specifically, when the value size is 16KB, the DRAM cost is reduced from more than 20GB to less than 5GB per node. When the value size is 64KB, 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.

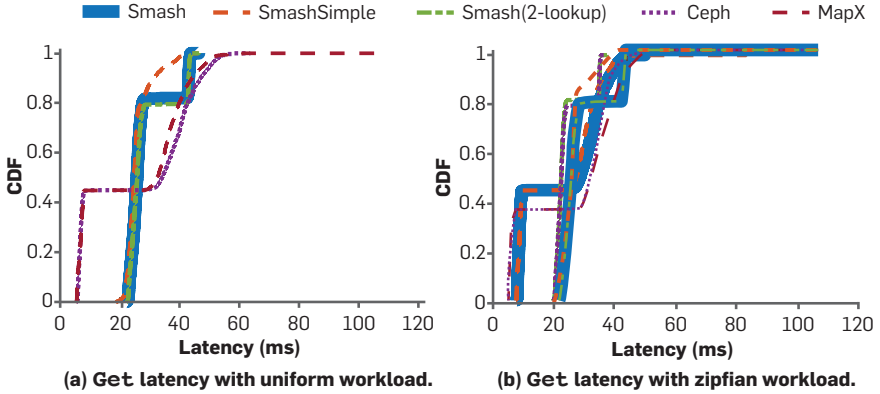
5.3. Latency of storage operations.

In this subsection, we show testbed evaluation results of storage-operation latency, including `Put` and `Get` under different distribution workloads. We first put 10,000 objects in the cluster, and then the client issues 4,000 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 running one and two lookup units respectively.

`Get`. Figure 7 shows the CDF of `Get` latency under the uniform (Figure 7a) and Zipfian (Figure 7b) workloads. 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` latencies. 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 a 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.

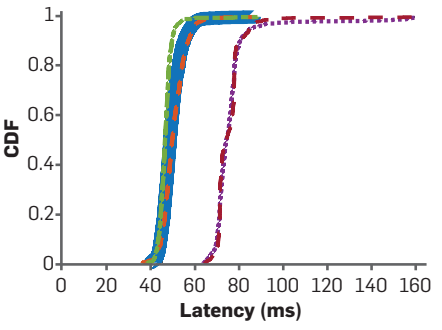
`Put`. Figure 8 shows the cumulative distribution (CDF)

Figure 7. Latency for getting objects.



(a) `Get` latency with uniform workload. (b) `Get` latency with zipfian workload.

Figure 8. Latency for `Put` latency w/uniform workload.



of Put latency for these three methods under the uniform workload.

6. RELATED WORK

Object storage[20] is a type of storage system that manages 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,[10,13] 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; these servers are called metadata servers in many file systems. Some object storage systems also apply this approach. Amazon S3[1] allows each user to put its objects into a bucket and maintain the full object-to-bucket mappings. For large-scale object storage, the resource overhead for the directory is huge and hard to replicate to avoid becoming a single point of failure. There are two main reasons: The number of objects is big, and the size of each key is also long, even on the same scale as the object.[2,24] Hence storing key-location mappings in the directory would cost a massive amount of DRAM space (for example, greater than 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. Hence, hashing fails to meet these application requirements. CRUSH[21] is the hashing-based placement-and-lookup method used in Ceph,[7,20] 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.

7. CONCLUSION

This paper presented 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. This allows the system to optimize object locations based on application requirements. Smash needs very little DRAM resources, and the per-node DRAM cost is lower than that of CRUSH and MapX. We implemented Smash using a testbed running in a public cloud and demonstrated its advantages by comparing it with existing work. Our future work will focus on applying Smash in other scenarios, such as edge computing.

8. ACKNOWLEDGMENTS

Y. Liu and C. Qian were partially supported by National Science Foundation Grants 2322919, 2420632, and 2426031. 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. Amazon Web Services. Amazon simple storage service. 2021; https://aws.amazon.com/s3/.
2. Atikoglu, B. et al. Workload analysis of a large-scale key-value store. In *In Proceedings of ACM SIGMETRICS* (2012).
3. Barcelona-Pons, D. et al. On the FaaS Track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th Intern. Middleware Conf.* (2019), 41–54.
4. Bel, O. et al. Geomancy: Automated performance enhancement through data layout optimization. In *Proceedings of 2020 IEEE Intern. Symp. on Performance Analysis of Systems and Software.*
5. 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 *Proceedings of ACM-SIAM SODA* (2007).
6. Cao, Z., Dong, S., Vemuri, S., and Du, D.H. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conf. on File and Storage Technologies* (2020), 209–223.
7. Ceph; https://docs.ceph.com/.
8. CloudLab; https://www.cloudlab.us/.
9. Fernholz, D. and Ramachandran, V. The *k*-orientability thresholds for $G_{n,p}$. In *Proceedings of ACM/SIAM SODA* (2007).
10. Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the 19th ACM Symp. on Operating Systems Principles* (2003), 29–43.
11. Karger, D. et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of ACM SOTC* (1997).
12. Klimovic, A. et al. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symp. on Operating Systems Design and Implementation* (2018), 427–444.
13. Li, S. et al. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the Intern. Conf. for High Performance Computing, Networking, Storage and Analysis* (2017), 1–12.
14. Li, X., Andersen, D., Kaminsky, M., and Freedman, M.J. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proceedings of ACM EuroSys* (2014).
15. Liu, Y. et al. Smash: Flexible, fast, and resource-efficient placement and lookup of distributed storage. In *Proceedings of the ACM on measurement and analysis of computing systems 7*, 2 (2023), 1–22.
16. MongoDB; https://github.com/mongodb/mongo.
17. Shi, S. and Qian, C. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems 4*, 2 (2020), 1–32.
18. Stoica, I. et al. Chord: A scalable peer-to-peer lookup service for Internet applications. *ACM SIGCOMM Computer Communication Rev. 31*, 4 (2001), 149–160.
19. Wang, L., Zhang, Y., Xu, J., and Xue, G. MAPX: Controlled data migration in the expansion of decentralized object-based storage systems. In *Proceedings of the18th USENIX Conf. on File and Storage Technologies* (2020), 1–11.
20. Weil, S.A. et al. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symp. on Operating Systems Design and Implementation* (2006), 307–320.
21. 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 Conf. on Supercomputing*, IEEE, 2006, 31–31.
22. won You, G., won Hwang, S., and Jain, N. Scalable load balancing in cluster storage systems. In *In Proceedings of the ACM/IFIP/USENIX Middleware Conf.* (2011).
23. Xie, M. and Qian, C. Reflex4arm: Supporting 100gbe flash storage disaggregation on arm soc. In *Proceedings of the OCP Future Technology Symp.* (2020).
24. 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).
25. 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).

Yi Liu, University of California Santa Cruz, CA, USA.

Shouqian Shi, University of California Santa Cruz, CA, USA.

Minghao Xie, University of California Santa Cruz, CA, USA.

Heiner Litz, University of California Santa Cruz, CA, USA.

Chen Chi, University of California Santa Cruz, CA, USA.