MOS: Workload-aware Elasticity for Cloud Object Stores

Ali Anwar, Yue Cheng
Virginia Tech
Blacksburg, VA
{ali,yuec}@cs.vt.edu

Aayush Gupta
IBM Research – Almaden
San Jose, CA
guptaaa@us.ibm.com

Ali R. Butt
Virginia Tech
Blacksburg, VA
butta@cs.vt.edu

ABSTRACT

The use of cloud object stores has been growing rapidly in recent years as they combine key advantages such as HTTP-based RESTful APIs, high availability, elasticity with a "pay-as-you-go" pricing model that allows applications to scale as needed. The current practice is to either use a single set of configuration parameters or rely on statically configured storage policies for a cloud object store deployment, even when the store is used to support different types of applications with evolving requirements. This crucial mismatch between the different applications requirements and capabilities of the object store is problematic and should be addressed to achieve high efficiency and performance.

In this paper, we propose MOS, a Micro Object Storage architecture, which supports independently configured microstores each tuned dynamically to the needs of a particular type of workload. We also design an enhancement, MOS$_{++}$, that extends MOS's capabilities through fine-grained resource management to effectively meet the tenants' SLAs while maximizing resource efficiency. We have implemented a prototype of MOS$_{++}$ in OpenStack Swift using Docker containers. Our evaluation shows that MOS$_{++}$ can effectively support heterogeneous workloads across multiple tenants. Compared to default and statically configured object store setups, for a two-tenant setup, MOS$_{++}$ improves the sustained access bandwidth by up to 79% for a large-object workload, while reducing the 95th percentile latency by up to 70.2% for a small-object workload.

Keywords

Object store; Performance analysis; Resource management and scheduling

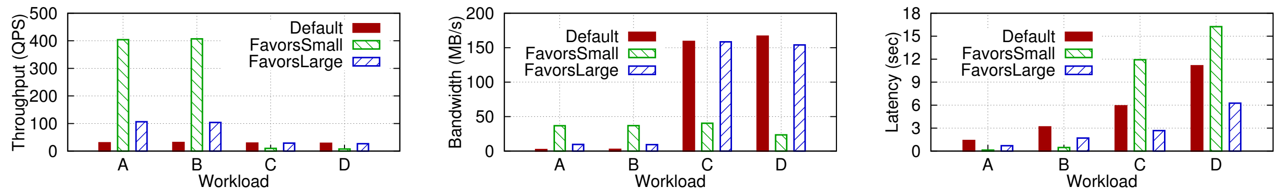
1. INTRODUCTION

Cloud object stores, such as Amazon S3, Google Cloud Store (GCS), OpenStack Swift and Ceph [34], have become the most widely used form of cloud storage in recent years. These stores combine key advantages such as high availability, elasticity and a "pay-as-you-go" pricing model, which allows applications to scale as the usage increases or decreases, and offers HTTP-based RESTful APIs for easy data management. The desirable features, coupled with the advances in virtualization infrastructure, are driving the adoption of cloud object stores by a myriad of applications. Examples range from web applications [14] that store image and video files, to backup services [3] that require large capacity for archival data, to big data analytics frameworks [17]. Similarly, object stores are increasingly being adopted by the HPC community [23] as they provide efficient metadata management and scalability that helps in extreme-scale high-end computing, and allows for seamless adaptation to a wide range of general purpose and scientific computing file system workloads [34].

A typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [10] with a static configuration to handle different types of applications with evolving requirements. Using a monolithic configuration setup results in all applications experiencing the same service level, e.g., similar average latency per request, data transfer throughput, and queries per second (QPS). However, different applications entail extremely different latency and throughput requirements. For example, a social networking or photo sharing application requires low latency to support a highly-responsive user experience, whereas backup services can tolerate higher latency but require sustained high throughput. Some object stores provide static configuration of storage policies, e.g., Swift segmentation [10], to allow for segmenting the cluster. This static storage segmentation policy is limited to cover only the storage server layer of an object store. However, a typical object store setup also consists of additional layers such as proxy server and load balancer layer. End-to-end performance of an application depends on correct configuration at all these layers to meet its requirements, and not only the storage server. Thus, a comprehensive solution is needed.

From the cloud provider's perspective, supporting dramatically different workloads from different applications (tenants) using a single homogeneous configuration means that optimization opportunities are lost. Each different application represents a workload with different characteristics. For example, a photo sharing application such as Instagramwould have a large number of small-medium sized files (e.g, KB- to MB-level image objects), with skewed access pattern where frequent read and write requests go for hotter/popular objects. In contrast, an enterprise backup application (e.g., Arq [1]) consists largely of write requests for large cold archive files with reads only sparsely arising. Us-

HPDC'16, May 31-June 04, 2016, Kyoto, Japan
© 2016 ACM. ISBN 978-1-4503-4314-5/16/05. . . $15.00
DOI: http://dx.doi.org/10.1145/2907294.2907304



(a) Throughput (QPS) (b) Bandwidth (MB/s) (c) Latency (s)

Figure 1: Performance achieved under various object store configurations in a multi-tenant environment.

ing a homogeneous configuration prevents fine-tuning of the system to such varied needs and reduces overall system efficiency.

The situation is further complicated by the fact that due to regular system upgrades and introduction of new storage architectures, data centers hosting the object stores are becoming increasingly heterogeneous [21, 30]. However, with either the "one-size-fits-all" monolithic deployment or static storage segmentation policy driven partitioning, it is impossible to match specific types of hardware with the right type of application workload. For example, latency-sensitive small-object workloads would require low-latency storage devices and powerful CPU processing capacity, whereas large object write-only workloads can be supported with a combination of high network bandwidth across all layers (e.g., load balancer, proxy, and object servers etc.) and weaker CPU power. Under these scenarios, meeting SLA requirement for one of the workloads may require, (i) adding hardware resources that may not improve the performance for other workloads, and (ii) software tuning that may decrease the performance for other workloads.

Furthermore, the workloads seen by the object store are varied and fluctuate over time. Consider a scenario where the workload demand from one application (tenant) is spiking while the demand from another application that shares the same object store resources is dipping. In this situation, static policies need to be updated based on the changes experienced in the workload. This calls for a new object store architecture that can dynamically perform resource provisioning for driving online reconfiguration across multiple partitions of the object store.

Hence in this paper, we posit that compared to using a rigid object store *it is more beneficial to support multi-tenant workloads separately using dynamically configurable finer-grained object stores* on sub-clusters of available resources.

Motivational Study. To motivate our approach and demonstrate the need for differentiated object stores, we study different types of representative practical workloads as follows. We examine four different real-world applications that use cloud object storage as listed in Table 1. We deploy and evaluate OpenStack Swift in a multi-tenant environment using COSBench [35] as workload generator configured for the four types of studied workloads. Swift is a popular object store implementation provided by OpenStack that is increasingly becoming the *de facto* cloud computing software platform. In these tests, we use three different Swift configurations (setups)[1]. We run COSBench clients on designated machines to saturate Swift. Each benchmark is run for 15 minutes after

Workload	Workload characteristics		App. scenario
	Obj. size	Operation distribution	
A	1–128KB	G: 90%, P: 5%, D:5%	Web hosting
B	1–128KB	G: 5%, P: 90%, D:5%	Online game hosting
C	1–128MB	G: 90%, P: 5%, D:5%	Online video sharing
D	1–128MB	G: 5%, P: 90%, D:5%	Enterprise backup

Table 1: Different types of workloads and application scenarios used for testing the behavior of object stores. `G`: GET operation; `P`: PUT operation; `D`: DELETE operation.

all data is loaded into the store. We use two nodes as proxy servers in each of the configuration. To simulate datacenter heterogeneity, one of the proxy server has 32 cores while the other has 8 cores. The proxy server running on the 32-core machine is connected to the storage nodes via 1 Gbps interconnect, while the proxy server on the 8-core machine is connected via 10 Gbps interconnect. In addition, four 32-core machines are used as storage nodes. Each storage node has 3 SATA SSDs. The storage nodes are well-endowed and configured in such a way so as not to become a performance bottleneck for any of the studied configurations.

Default configuration: The default monolithic Swift setup is used where both 8-core and 32-core machines acted as proxy server. The workloads are handled by all resources and round robin DNS was used to distribute the requests to the proxies.

FavorsSmall configuration: The available resources are divided into two sub-object stores, one configured for workloads with small objects and the other for large objects. One 8-core machine (connected via 10 Gbps) served as proxy for `WorkloadA` and `WorkloadB`, and one 32-core machine connected via 1 Gbps network served `WorkloadC` and `WorkloadD`.

FavorsLarge configuration: One 32-core machine (connected via 1 Gbps) is used as proxy for `WorkloadA` and `WorkloadB`, while one 8-core machine (connected via 10 Gbps) is used as proxy for `WorkloadC` and `WorkloadD`.

Figure 1 shows the comparison of performance achieved under the studied configurations. As shown in Figure 1(a), separating proxy servers for different workloads improved the overall QPS by 700% and 225% for `FavorsSmall` and `FavorsLarge`, respectively, compared to the default Swift setup. It is interesting to note that even though `FavorsSmall` resulted in very high QPS for small objects of (`WorkloadA` and `WorkloadB`), it is not the best configuration as it significantly affects the MB/s (dropped by from 350% to 500%, as observed in Figure 1(b)) for workloads dominated with large object (`WorkloadC` and `WorkloadD`). On the other hand, in `FavorsLarge` the throughput for large objects remained same.

[1]The integrated Swift storage policies only support static storage node segmentation. In our motivational study we keep the storage node settings fixed and vary the proxy node settings. We do this to highlight the need for a more comprehensive workload-aware scheme, which is the subject of this paper.

Similarly, the latency of `FavorsLarge` is also less than that achieved by the default configuration for all the workloads (Figure 1(c)). `FavorsSmall` provides best and worst latency for small and large object workloads, respectively. We also observe that switching to different network connections on proxy servers in `Default` configuration results in similar results. These results demonstrate the need for a comprehensive study of the impact of different configurations on performance to ensure efficient cloud object store design.

From our experiments, we infer the following. (i) Cloud object store workloads can be classified based on the size of the objects in their workloads. In case of small objects, cloud tenants are mostly interested in QPS and latency, whereas for large objects data throughput is considered more important. (ii) When multiple tenants run workloads with drastically different behaviors, they compete for the object store resources with each other, the workload dominated with small objects experiences a dramatic loss in performance. This is because the available network bandwidth is exhausted to transfer TCP packets containing payload for large objects, hence wasting the CPU power that would have been utilized to serve workloads with small objects on object storage nodes. That is why using a separate proxy server under `FavorsSmall` and `FavorsLarge` gives a fair chance to small object workloads to be properly handled by the storage nodes. Thus, cloud object stores need better resource management to ensure that tenants are treated equally.

Contributions. To this end, we propose MOS, a novel micro object storage architecture with independently configured micro-object-stores each tuned dynamically for a particular type of workload. We then expose these microstores to the tenants who can then choose to place their data in the appropriate microstore based on the latency and throughput requirements of their workloads. We further enhance our basic resource provisioning engine to build MOS$_{++}$, which incorporates the container abstraction for fine-grained resource management, SLA awareness, and better resource efficiency.

Specifically, we make the following contributions:

- We evaluate the impact of conventional object storage configuration on performance and resource efficiency by conducting experiments on a local Swift testbed. Our observations stress the need to carefully evaluate the various configuration choices and develop simple *Rules-of-Thumb* that cloud providers can leverage for provisioning the object storage.
- We perform a detailed performance and resource efficiency analysis on identifying major hardware and software configuration opportunities that can be used to fine-tune object stores for specific workloads. Our findings indicate the need to re-architect cloud object storage specialized for the public cloud.
- Based on our behavior analysis, we design MOS, an object store that (i) dynamically provisions microstores, each configured with different combination of hardware and software options, and (ii) exposes the interfaces of microstores to the tenants to use according to application requirements.
- We extend our basic framework to MOS$_{++}$ that uses container based approach to launch resources in a more fine-grained manner. MOS$_{++}$ is SLA-aware, supports rapid deployment, portability across machines, offers a lightweight footprint, and simplifies maintenance.
- We implement a prototype of MOS$_{++}$, and demonstrate that our approach results in improved performance (by up to 89.6% and 79.8% compared to the default monolithic and statically configured object store setup, respectively), as well as higher resource efficiency. Furthermore, we design a simulator to evaluate our solution under a large-scale 456-core cloud cluster setup. We also compare the performance of MOS with MOS$_{++}$ to highlight advantages of our container based approach.

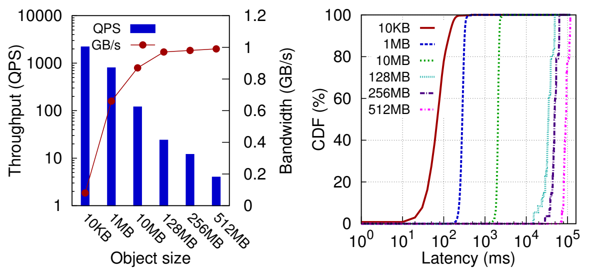
2. BACKGROUND AND RELATED WORK

Object Store Segmentation. Swift provides storage policies to support for segmenting the cluster through the creation of multiple object rings [10]. This feature is useful if a provider wants to offer different level of durability, performance, or storage implementation but does not want to maintain separate clusters. In contrast, MOS advocates to separately maintain clusters to incorporate segmentation across all layers. Furthermore, storage policies are static whereas MOS dynamically perform resource provisioning that can drive online reconfiguration across multiple partitions of the object store. Also, performance comparison of the Swift-based prototype of MOS with other object stores like Ceph will not be an apple-to-apple comparison as Ceph outperforms Swift [2] and they have significantly different architecture.

Workload-aware Elasticity. The focus of various recent research works have been on providing an elastic setup for cloud based storage. Lim et al. [29] propose an elastic storage system on HDFS for multi-tier application services. Similarly, ElasTraS [20] provides scalability and elasticity to the data store in clouds for optimizing transactional data access. MeT [19] focuses on systems metrics (CPU utilization, I/O wait and memory usage) that are critical for a NoSQL database. Skute [15] provides a fault-tolerance and scalable replication scheme for cloud storage. MOS differs from these works in that it focuses on providing best performance guarantee for heterogeneous multi-tenant workloads by exploiting automated elasticity for cloud object store.

Handling Cluster/Workload Heterogeneity. hatS [26] proposes a replication scheme for HDFS that integrates heterogeneous storage technologies into Hadoop. ϕSched [27] designs a cluster-heterogeneity-aware scheduler to improve the resource-application match. Walnut [16] suggests using a hybrid object strategy to support both small and large objects in an object store. CAST [17] and its extension [18] perform coarse-grained cloud storage (including object stores) management for data analytics workloads. In contrast, MOS explicitly partitions the conventional monolithic storage into multiple dynamically tuned microstores, each serving a particular type of workload.

Meeting SLA/SLO. SCADS [33] uses a steady-state performance model to predict whether a server can handle a particular workload, without violating a given latency threshold. SCADS reconfigures the storage system on-the-fly in response to workload changes driven by a performance model. Similarly, Papio [32] introduces a QoS-enabled function into the S3-based object store where it accepts an explicit performance request as an advanced reservation, and enables QoS in the access with the extended S3 RESTful interfaces. MOS differs from these works in that it: i) keeps track of



(a) Read performance. (b) Read latency distribution.

Figure 2: Impact of varying the object size on read performance. Note the log scale on the Y-axis of Figure 2(a) and the X-axis of Figure 2(b).

fine-grained resource usage; and ii) partitions the heterogeneous workloads and optimizes each individually based on tenants' SLA requirements while yielding higher overall performance and better resource efficiency.

Dynamic Resource Management. Mantle [31] is a programmable storage system that lets users inject custom balancing logic into Ceph [34]. This feature provides flexibility for allocating resources. Unlike Mantle, MOS automatically performs resource provisioning without burgeoning the users. [13] propose a fine-grained resource allocation mechanism based on metrics such as CPU utilization, I/O wait and memory usage that are critical for MapReduce workloads. Similarly, Lee et al. [28] design a heterogeneity-aware resource allocator and job scheduler for a cloud data analytics system. While these works provide heterogeneity-aware optimizations targeted at MapReduce workloads, MOS targets object stores and focuses on improving the performance of real-time request processing.

3. ANALYSIS

In this section, we present a detailed analysis of how object store behaves under various software and hardware configurations. Next, we use the study to develop *rules-of-thumb* for configuring object stores, which guide the design of MOS.

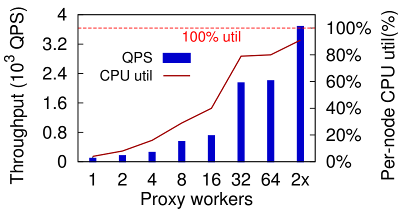
In the following analysis, we use a 32-core machine as a proxy node with two 32-core storage nodes each equipped with 3 SSDs (to eliminate the storage bottleneck), unless mentioned otherwise. For workloads dominated by small objects (at KB level) the metrics of interest are *throughput* in terms of queries per second (QPS) and response latency, while for workloads dominated by large objects (at MB–GB level), *bandwidth* in terms of MB/s or GB/s is more important.

Q1: How does object size impact performance? First, we analyze the impact of object size on performance in terms of throughput (QPS) and bandwidth (GB/s). While QPS captures the object-wise throughput performance, the bandwidth serves as an important metric reflecting byte-wise performance. As shown in Figure 2(a), increasing the object size results in the throughput decreasing drastically. Specifically, when the object size is increased from 10 KB to 10 MB, we observe the increasing tendency of the network bandwidth. When the object size is increased further to above 128 MB, the bandwidth only improves marginally (from 0.97 GB/s to 0.98 GB/s), implying that the NIC is saturated. Figure 2(b) plots the corresponding latency distribution at each studied object size. At large object sizes (10 MB–512 MB), the request response latency is more than 100× than that for small object sizes (10 KB–1 MB). From these tests, we can infer that, as long as the object size exceeds a certain threshold, network bandwidth becomes the limiting factor. Correspondingly, this again, explains why `WorkloadA` and `WorkloadB` achieve extremely poor performance when co-existing with `WorkloadC` and `WorkloadD` in §1. Hence, the tests demonstrate that, in a multi-tenant environment with mixed workloads, individual workloads should be partitioned and serviced through disjoint object stores to reduce mutual interference and performance impact.

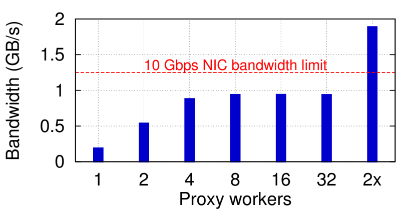
Q2: How does proxy server configuration impact performance? Next, we study the effect of scaling proxy nodes on workload performance. We vary the computational capacity of the proxy node by increasing proxy's allotted CPU cores. Figure 3(a) shows the proxy tuning effect. As we increase the proxy workers in one proxy node the QPS is improved linearly until we reach 32 proxy workers. The observed CPU utilization reaches close to 85% (bounding the throughput) with both 32 and 64 proxy workers, implying that CPU becomes the bottleneck here. Adding one more proxy node (`2x`) almost doubles the performance (QPS increased from 2, 200 to 3, 700), clearly demonstrating that proxy's performance is constrained by the CPU capacity. Next, we repeat the test with large object workloads. As shown in Figure 3(b), the network bandwidth limit is reached as soon as the number of proxy workers reaches 4, with modest CPU utilization (about 25%) observed on the proxy node. This is because for large object workload, the performance becomes constrained by the network bandwidth before CPU can be saturated. Hence adding another proxy node (`2x`, i.e., doubling the available network bandwidth) results in linear increase in throughput. Thus, the takeaway is that a proxy's computational capacity can act as the bottleneck for workloads dominated with small objects, whereas the network bandwidth is the limiting resource for workloads dominated by large objects.

Q3: How does storage server configuration impact performance? Next, we study the effect of scaling object storage nodes on workload performance. As shown in Figure 3(c), the peak QPS for small object workloads is achieved with 16 object storage workers, which is exactly the same as the number of proxy workers launched to achieve this QPS (recall that two object storage nodes are deployed behind one proxy server node). This implies that the maximum performance can only be achieved when both the proxy and storage nodes are equipped with the same amount of CPU resources, which strengthens our observation that CPU capability is the limiting factor for small-object workloads. In contrast, for large-object workloads, the network limit is quickly reached with only 4 object storage workers. This is because, for large objects the performance is bottlenecked by the network (recall that each storage node has 3 SATA SSDs, thus disk bandwidth does not pose a limitation in our test).

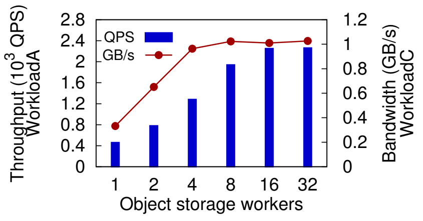
Q4: How does network/storage affect performance? In our next test, we study the effect of varying storage device and network connectivity on workload throughput. Figure 4(a) shows that faster network interconnect (1 Gbps NIC → 10 Gbps NIC) results in only 12% increase in QPS for small object workloads with HDD as storage medium, and 70% increase when SATA SSD is used. This observation shows that small-object intensive workloads are more sensitive to the storage devices rather than the network



(a) Effect of varying proxy capability (A).

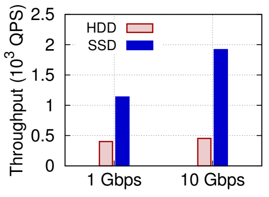


(b) Effect of varying proxy capability (C).

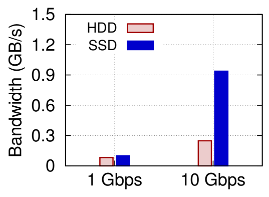


(c) Effect of varying object store server capability.

Figure 3: Studied software/hardware configuration options. In Figure 3(c), small-object workloads refer to bars (QPS) while large-object workloads refer to linepoints (GB/s). A: `WorkloadA`; C: `WorkloadC`.



(a) `WorkloadA`.



(b) `WorkloadC`.

Figure 4: Performance of the object store equipped with **homogeneous storage devices** as a function of the NIC bandwidth.

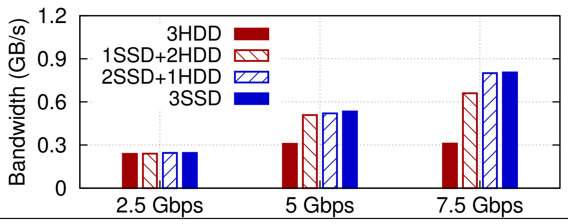


Figure 5: Performance of large-object workload with **heterogeneous storage devices** as a function of the NIC bandwidth. The corresponding bandwidth (GB/s) with 2.5 Gbps, 5 Gbps, and 7.5 Gbps are 0.31 GB/s, 0.62 GB/s, and 0.93 GB/s, respectively.

bandwidth. Thus, they may be efficiently handled using a lower-bandwidth network interconnect but by using high-bandwidth storage devices. On the other hand, increasing network interconnect improves performance by as much as 900% (using SSDs) in case for large-object intensive workloads (Figure 4(b)), which clearly indicates such kind of workloads can benefit from high-bandwidth network interconnects.

Q5: What is the impact of heterogeneous storage setup on performance of large-object workloads? Finally, we study the impact of heterogeneous storage configuration on large-object intensive workloads. Here, we limit the network bandwidth using Linux traffic control tool `tc`. We measure the performance of the object store under a large-object workload, with four setups: two heterogeneous setups 1 SSD + 2 HDD and 2 SSD + 1 HDD; and two homogeneous setups 3 HDD and 3 SSD as baselines. Figure 5 demonstrates that the choice of different storage device type combination changes based on the network bandwidth limit. We vary the network bandwidth limit to emulate the scenario where the network is partitioned in a multi-tenant environment. Note, when the network is limited to 2.5 Gbps, all four storage configurations achieve the same performance. Thus, the storage setup of choice under 2.5 Gbps is 3 HDDs. As the bandwidth limits increases to 7.5 Gbps, the 3 HDDs setup becomes the worst choice, especially when meeting SLAs is critical. Here, the 2 SSD + 1 HDD setup is desirable as it achieves almost the same performance as the 3 SSD setup, but with a higher resource efficiency. These tests necessitate the need for a workload-aware resource provisioning mechanism that selects the most efficient and high performing options under dynamically changing workloads and tenant requirements.

Summary of Rules-of-thumb. It is fairly straightforward to manually tune the object stores by controlling all the other configuration variables. However, it is a challenging task to dynamically detect the workload shifts and meet the tenant goals while maximizing the resource efficiency at runtime, particularly when the service providers are faced with many software and hardware configuration options. To this end, we develop the following *rules-of-thumb* that are helpful in guiding the online/offline performance tuning of object stores as well as the design of MOS.

R1 Cloud object store design can benefit from (i) partitioning the monolithic object store architecture based on workload characteristics, and (ii) separately servicing interfering workloads in the multi-tenant environment. Object size distribution is a key factor for classifying workload characteristics.

R2 CPU capacity of proxy servers is the first-priority resource for small-object intensive workloads. CPU becomes a bottleneck much earlier than the network for such workloads.

R3 On the other hand, availabile network bandwidth plays a critical role in the performance of large-object intensive workloads.

R4 The number of CPU cores used in storage nodes can be safely configured based on the number of deployed proxy workers, given that the storage devices provide sufficient disk bandwidth. This rule can be modeled using the following equation: $proxyCores = storageNodes{*}coresPerStorageNode$. E.g., one 32-core proxy node may require four 8-core storage nodes.

R5 The aggregated network bandwidth between proxy and storage nodes should be roughly the same as the link bandwidth used by cloud provider to connect to the proxies. Generally, this rule can be modeled as: $bw_{proxies} = storageNodes{*}bw_{storageNode}$.

R6 A faster network cannot effectively improve QPS for small-object intensive workloads. For tenants who do not impose strict SLO requirements, the workload, if dominated with small objects, may be better served using a combination of low-bandwidth network (i.e., 1 Gbps NICs) with high-bandwidth storage devices (e.g., SSD delivering decent random and sequential I/O performance). This low-cost het-

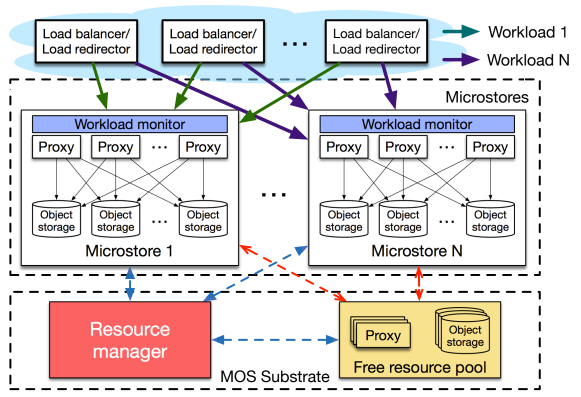


Figure 6: Overview of MOS architecture.

erogeneous resource combination can effectively meet tenants' requirement while improving data center cost efficiency.

R7 For large-object intensive workloads, we have to collectively consider the network bandwidth limits and the storage configuration. Given a certain network limit and SLA, a combination of slow and fast storage devices (e.g., HDD+SSD) may be able to serve the application needs in a resource efficient manner.

4. MOS FRAMEWORK

We design MOS based on the *rules-of-thumb* developed in §3, with the goal to address workload and datacenter resource heterogeneity. Instead of a conventional monolithic storage architecture, where all tenants/workloads share the storage resources using same or static configurations, MOS uses multiple object stores, which we refer to as microstores. To service changing workloads, MOS performs dynamic resource partitioning and provisioning, allowing each microstore within an object storage setup to run as a fully-functional object store unit. As pointed in our earlier motivational study, although the monolithic approach is simple to implement and configure, it is not necessarily resource efficient and can lead to wastage of CPU etc. resources.

Figure 6 shows the architecture of MOS, which consists of two layers: (1) **Microstores:** consists of multiple instances of object stores, a *microstore*, which is allocated a subset of proxy nodes and storage nodes that matches the requirements of application the microstore will support. (2) **MOS substrate:** consists of a resource manager that monitors the load on each microstore using a workload monitor and automatically reconfigures the resources assigned to the microstore to cope with workload shifts.

Microstores. A typical MOS deployment configures multiple instances of object store. Each instance comprises a subset of nodes in the cluster, and constitutes a unique microstore. Each microstore consists of two types of nodes: proxy nodes and storage nodes (each statically equipped with *hardware resources* such as CPUs, network interface cards (NICs) and storage devices, etc.). The proxy nodes are responsible for directing queries and managing the metadata for key space and replicas, etc. Whereas, the storage nodes store the object data in their local file systems and replicate the objects at multiple storage nodes for fault tolerance. The number of microstores configured in a deployment of MOS is specified by the service provider and will depend on the kinds of workloads that need to be supported. At any given time during MOS's operation, the amount of resources allocated to each microstore is determined by its resource utilization. Such elasticity guarantees resource efficiency while delivering the needed performance.

MOS substrate. The MOS substrate layer consists of a central *Resource Manager* and the same number of *Workload Monitors* as the number of microstores. The main function of this layer is to perform online performance and resource utilization monitoring. The workload monitors are used to gather statistics of each running microstore, which are polled periodically by the resource manager to decide when and if a reconfiguration is warranted. The workload monitors also execute, as needed, online microstore reconfiguration commands issued by the Resource manager to meet tenants' demands. The resource manager manages the available pool of heterogeneous resources. It makes decisions about when and how to add or redistribute resources for the reconfiguration of a microstore in order to respond to workload shifts in the observed throughput and latency. We design the MOS framework in a modular fashion, where the core resource provisioning mechanism is configurable depending on tenants' needs.

Algorithm 1: MOS Resource Provisioning Algorithm.

```
Input: microstores: Microstore array, free_pool: free
       resource pool, $util_{low}$: low utilization threshold,
       $util_{high}$: high utilization threshold, epoch:
       configurable monitoring interval
begin
    $microstores.hw \leftarrow$ init($free\_pool$)
    while true do
        foreach ms in microstores do
                // periodically collect monitoring stats
            if $util_{low} \leq$ util($ms.hw$) $\leq util_{high}$ then
                $ms.firstTime \leftarrow true$
                continue
            else
                if $ms.firstTime$ then
                    $ms.firstTime \leftarrow false$
                    $ms.toChange \leftarrow 1$
                else
                    $ms.toChange \leftarrow ms.toChange * 2$
                if util($ms.hw$) $> util_{high}$ then
                            // to add in more resources
                    $ms.hw \leftarrow ms.hw + ms.toChange$
                    /* allocate resource from free resource pool */
                    alloc($free\_pool$, $ms$, $ms.toChange$)
                else if util($ms.hw$) $< util_{low}$ then
                            // to remove resources
                    $ms.hw \leftarrow ms.hw - 1$
                    /* return resource to free pool */
                    dealloc($free\_pool$, $ms$, 1)
        sleep($epoch$)
```

5. BASIC MOS

We first design a basic version of MOS. The goal of the basic version is to provide a coarse-grained dynamic resource management mechanism that can help achieve good resource utilization without worrying about SLA enforcement. The core of the basic MOS uses a greedy heuristic to perform resource provisioning based on online workload changes.

Basic MOS Algorithm. Algorithm 1 takes as input *microstores*, a vector of all microstores storing statistics such as hardware configuration, current load being served, and the resource utilization, e.g., CPU and network bandwidth utilization. Initially, the algorithm allocates the same amount of resources to each microstore conservatively. It then enters into the main loop, where the resource manager periodically (with configurable *epochs*) polls each microstore. In each iteration, if the resource utilization (fetched using `util`($ms.hw$)) of one microstore lies within a pre-defined threshold range (i.e., $[util_{low}\ util_{high}]$), the algorithm simply moves to the next microstore. If the microstore is in sub-optimal state, the algorithm decides to *quadratically* add or *linearly* remove resources. This is to ensure that the algorithm will not overshoot the de-allocation of resources, but can quickly respond to sudden workload increases.

6. ENHANCEMENTS: MOS++

There are two limitations of the basic MOS resource provisioning algorithm: (i) it is not SLA-aware; and (ii) it lacks the ability to perform fine-grained resource management. We address these shortcomings by designing support for container based deployment for object store in MOS to create MOS++. Containers greatly improve the flexibility for dynamic reconfiguration. For example, by leveraging containers, MOS++ can specify the number of CPU cores added to each microstore, support utilizing different types of storage devices in various configurations, and perform network partitioning.

Algorithm 2: MOS++ Resource Provisioning Algorithm (SLA-aware and Container-based).

```
Input: $free\_pool$: free resource pool, $sla_{low}$: low SLA
       threshold, $sla_{high}$: high SLA threshold, $util_{low}$:
       low utilization threshold, $util_{high}$: high
       utilization threshold, $util_{thresh}$: % amount of
       time sampled in one epoch, $epoch$: configurable
       monitoring interval
begin
    $ms \leftarrow$ init($sla$, $free\_pool$)
    while $true$ do
        if $sla_{low} \leq ms.perf \leq sla_{high}$ then
            if $util_{low} \leq ms.util \leq util_{high}$ then
                continue
            else
                // fine-tune the ms config
                if $ms.util > util_{high}$ then
                    $ms.cont \leftarrow ms.cont + 1$
                    /* allocate resource from free resource pool */
                    alloc($free\_pool$, $ms$, 1)
                else if $ms.util < util_{low}$ then
                    $ms.cont \leftarrow ms.cont - 1$
                    /* return resource to free pool */
                    dealloc($free\_pool$, $ms$, 1)
        else
            if $sample(util_{low} \leq ms.util \leq util_{high})$
            $> util_{thresh}$ then
                continue
            else if $ms.perf > sla_{high}$ then
                                        // to remove containers
                $toChange \leftarrow$
                getContainers($ms.perf - sla_{high}$)
                $ms.cont \leftarrow ms.cont - toChange$
                dealloc($free\_pool$, $ms$, $toChange$)
            else if $ms.perf < sla_{low}$ then
                                        // to add in containers
                $toChange \leftarrow$ getContainers($sla_{low} - ms.perf$)
                $ms.cont \leftarrow ms.cont + toChange$
                alloc($free\_pool$, $ms$, $toChange$)
        sleep($epoch$)
```

Specifying SLAs. We consider latency-based SLAs with constraints for small-object intensive workloads and bandwidth-based SLAs with constraints for large-object intensive workloads. Each SLA, which is associated with one object size attribute, has two parameters:

Notation	Description
E	Entities of the object store (proxy/object server)
cpu_{ij}^{E}	1 if CPU j is assigned to workload i, 0 otherwise
S	Disk storage type (PCIe/SATA SSD, HDD, etc.)
$disk_{ik}^{S}$	1 if Disk k is assigned to workload i, 0 otherwise
nw_i	Network bandwidth assigned to workload i

Table 2: Notations used in the LP model employed in MOS++.

- Average request response time (average latency) for small-object intensive workloads *OR* average bandwidth for large-object intensive workloads. For example, for a workload where 90% of all requests have a size of 10 KB , the average latency must be within a particular range, i.e., sla_{low} and sla_{high}, where sla_{low} is associated with the upper bound in terms of average latency and sla_{high} with the lower bound. Similarly, an SLA has a sla_{high} and sla_{low} associated with the upper and lower bound in terms of bandwidth, respectively, if the workload is large-object dominant, i.e., most objects with a size greater than 10 MB.

- Resource utilization: The fraction of time ($util_{thresh}$) the system resource utilization is within a specified range, e.g., $util_{low}$ and $util_{high}$.

The SLA requirement is met if *either one of the above two* parameters is true.

MOS++ Algorithm. Fine-grained resource allocation enables SLA-aware resource provisioning using Algorithm 2, which is an enhancement of Algorithm 1. In addition to the input parameters provided to the basic Algorithm 1, the enhanced algorithm takes three extra parameters: sla_{low}, low SLA threshold; sla_{high}, high SLA threshold; and $util_{thresh}$, fraction of time slots sampled during the period of one epoch.[2] In a workload, if the performance of a microstore goes above sla_{high}, resources are removed and reclaimed at per-container granularity from that microstore based on the suggested value provided by the function *getContainers*(.). Conversely, resources are allocated and added into the microstore whose performance is observed to go below the sla_{low} value. We set a precondition of resource utilization as the second parameter for our SLA. Function *sample*(.) periodically measures the resource utilization on a per-epoch basis. Again, the resources that need to be added are suggested by function *getContainers*(.), which uses a linear programming (LP) optimizer to compute a near-optimal allocation plan.

Resource Provisioning Optimization. We define the amount of resources allocated to workload i as:

$$resource_i = \sum_j cpu_{ij}^{E} + \sum_l disk_{il}^{S} + nw_i \ . \qquad (1)$$

The resource provisioning problem is modeled as an optimization problem to minimize the resources used for all workloads. Specifically, the objective is to:

[2]The sampling probability (%) is configurable. Each time slot is configured as 1% of the epoch. E.g., for an epoch of 5 minutes, a sampling time slot is 3 seconds.

$$minimize \sum_i resource_i \ , \tag{2}$$

$$s.t.\ \ sla_i^{low} \leq \text{cosperf}(resource_i) \leq sla_i^{high}, \forall i \ , \tag{3}$$

$$0 \leq \sum_j cpu_{ij}^E \leq 1, cpu_{ij}^E = 0 \text{ or } 1, \forall i \ , \tag{4}$$

$$0 \leq \sum_l disk_{il}^S \leq 1, disk_{il}^S = 0 \text{ or } 1, \forall i \ , \tag{5}$$

$$0 < \sum_i nw_i \leq nw_{max} \ . \tag{6}$$

Table 2 describes the notations used for representing the above model. Constraint 3 is used to guarantee that the SLA (the average response time for small-object workloads and the average bandwidth for large-object workloads) requirement is met. We profile the performance offline using an extensive stress test by iterating through nearly all possible resource configurations (§3). The estimated performance of a particular workload is estimated using function $cosperf(.)$, fed with the provisioned resources. Constraints 4 and 5 make sure that a CPU core or a storage drive can only be assigned to one workload. Similarly, Constraint 6 restricts the maximum network bandwidth (nw_{max}) that can be allocated to all the workloads. In the context of object stores, the CPU resources are allocated to two entities (denoted using E): proxies and object servers. While our model is general enough to cover many types of storage devices, we focus on three extant storage types (denoted using S) – PCIe/SATA SSDs, HDDs – in this work. The fairly small problem size implies that the optimization problem can be solved quickly, i.e., in seconds using CPLEX [8].

7. IMPLEMENTATION

Figure 7 shows the implementation details of MOS and MOS++. We build MOS on top of the Mesos [24] framework. The Mesos resource management is driven by the resource provisioning algorithms, i.e. Algorithm 1 or Algorithm 2. MOS launches Swift directly on the physical nodes serving as Mesos slaves. For MOS++ we extend Volt [11], a Mesos framework that can be used to launch containers on Mesos slaves.

For fine-grained resource allocation, we enforce the runtime constraints on resources while launching the Docker containers using the options provided by Docker [7]. The number of CPU cores is configurable for launching applications inside a container. Docker supports CPU core binding and disk size partitioning. For efficient disk utilization, we launch the object server containers with privileged mode, which enables the usage of devices attached to the host machine inside the container. We leverage this option to attach suitable storage devices for launched containers. To perform network partitioning, we use control groups (`cgroups`) [4] with Linux traffic control (`tc`) [9].

Container based deployment of Swift offers several advantages such as rapid deployment, portability across machines, easy sharing, lightweight footprint and simplified maintenance. All these advantages come at a cost of negligible or "close to zero" overhead [22]. To provide high availability, the Mesos cluster is launched with 3 masters with one acting as the leader and the rest on standby. In case of a master failure, a standby leader becomes the leader as a result of the election done by Zookeeper [25].

We have implemented our own proxy executor, object ex-

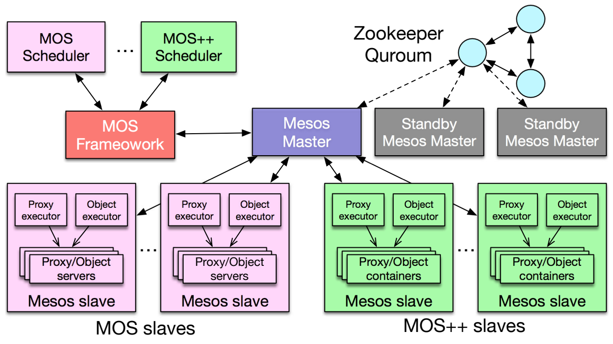


Figure 7: MOS modules and their interactions. Mesos provides support to launch Swift either directly on physical nodes or inside containers.

ecutor, and Python scripts to automate online reconfiguration of Swift in a distributed setup. The proxy executor is used to support dynamic reconfiguration of proxy servers inside the containers. The object executor performs online disk capacity management using the `swift-ring-builder` tool[3], whenever the storage configuration is updated. The code to build Docker container images and the Docker images for Swift proxy and object server are publicly available on Github [5, 6].

8. EVALUATION

We present the evaluation of MOS++ using both a prototype implementation and simulations. We first use the prototype to evaluate a number of object store setups under multi-tenancy in both static and dynamic workloads. This is followed by a simulation study of a large-scale system to compare MOS++ and MOS.

8.1 Prototype Evaluation

Experimental Methodology. We evaluate MOS++ using a 128-core local testbed. The testbed is connected using a 10 Gbps switch, with a maximum bandwidth of 40 Gbps. We emulate a two-tenant (client) environment, i.e., we run COSBench on two separate machines within the same subnet. We use `WorkloadA` (small-object read-intensive workload) and `WorkloadC` (large-object read-intensive workload) for this purpose. We compare MOS++ against two different object store setups – `Default`, where we use off-the-shelf monolithic configuration of Swift, and `Static`, where we statically configure two micro object stores designated for two tenants based on the rules-of-thumb of §3. The static approach is more advanced than the default segmentation policies [10] and serves as another point of comparison for our approach. Note that we focus on MOS++ for our prototype evaluation as it also encompass the basic design of MOS.

The `Default` setup is launched directly on the physical machines. `Static`, like MOS++, is launched inside containers. For `Static` setup, we tried several different overall configurations and selected the best one. Specifically, 75% CPU cores, 30% of NW bandwidth, and 100% PCIe SSD with 30% SATA SSD are assigned to `WorkloadA`. Accordingly, 25%

[3]Swift lazily migrates existing data replicas to newly added disks where new data can be instantly written and persisted. We leverage this feature to (i) control the I/O performance on new data that is written in newly added disks, and (ii) amortize the data migration cost.

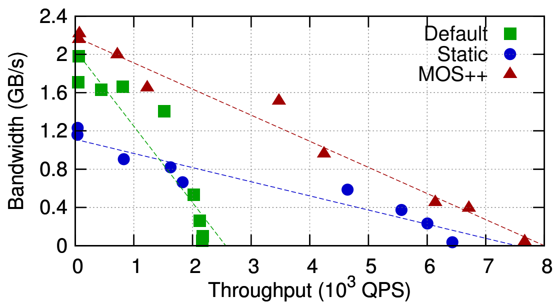


Figure 8: Overall throughput vs. bandwidth observed under different setups. Dotted lines are generated using linear regression, indicating the linear relationship between the overall throughput and bandwidth.

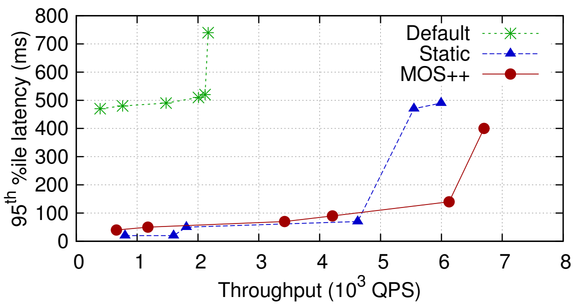


Figure 9: Throughput vs. 95^{th} percentile latency under `WorkloadA`.

CPU cores, 70% NW bandwidth, and 70% SATA SSD are assigned to `WorkloadC`. MOS++ starts initially with the same configuration as `Static` throughout our evaluation. Regarding runtime parameters, we set sla_{low} to be proportional to the workloads' load and sla_{high} $2\times$ of sla_{low}. We set $util_{low}$ 65% and $util_{high}$ 85%. We set *epoch* to be 3 minutes and $util_{thresh}$ as 80%.

Performance Evaluation. In our first set of experiments, we evaluate MOS++'s ability to handle heterogeneous varying workloads. We vary the COSBench processes from 2 to 1024 for `WorkloadA` to increase the throughput, while we decrease `WorkloadC`'s load by varying the COSBench processes from 32 to 2. Figure 8 plots the overall performance of the two studied workloads in terms of both throughput (QPS) and bandwidth (MB/s). `Default` achieves significantly higher bandwidth compared to `Static` when `WorkloadC` dominates (the far left part on X-axis dimension). This is because the large-object workload consumes most of the network bandwidth to transfer packets containing payload for large objects. Guided by our rules-of-thumb, `Static`'s statically provisioned micro store setup is able to balance the performance of both workloads to some extent. Hence, as `WorkloadA` gradually increases and eventually dominates, `Static` outperforms `Default` by as much as $2\times$. By leveraging workload-aware elasticity support, MOS++ combines the "best of both worlds", hence we see 10.4–89.6% improvement in overall throughput and 7.6–79.8% improvement in overall bandwidth, compared to both `Default` and `Static`. Thus, MOS++ is able to improve the overall performance for the two tenants with workloads exhibiting dramatically different characteristics.

Figure 9 depicts the 95^{th} percentile read tail latency and throughput tradeoffs observed for `WorkloadA`. For `WorkloadA`, `Default` performs the worst and lies in the upper-left corner

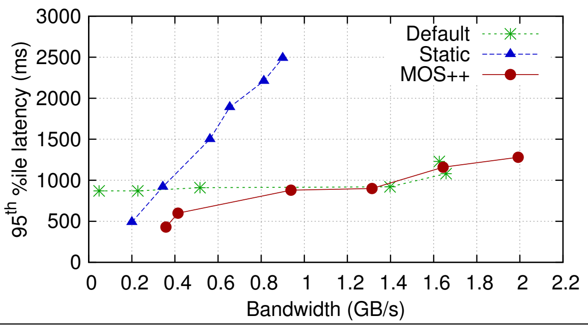


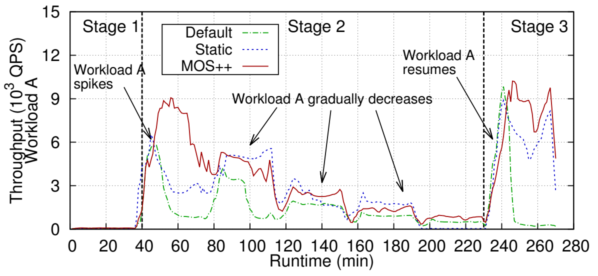
Figure 10: Bandwidth vs. 95^{th} percentile latency under `WorkloadC`. Average latency results show similar trend.

of the scatter chart. `Static` achieves comparatively similar performance with MOS++ as `WorkloadA` starts to increase. By adapting to the increasing load and adding more CPU power for `WorkloadA`, MOS++ eventually outperforms `Static` at peak loads (the right-most two data points) by up to 11.7% in throughput and up to 70.2% in tail latency. Figure 10 shows a similar trend under `WorkloadC`. Under the large-object dominant workload, `Static` is bottlenecked by its statically allocated network resource and hence limits the bandwidth for `WorkloadC`. Accordingly, we observe up to 79% improvement in bandwidth and up to 50.6% reduction in tail latency, compared to `Default`. Thus, MOS++ is able to improve the performance for both tenants, and effectively remove the performance bottleneck observed in `Default` and `Static` setups.

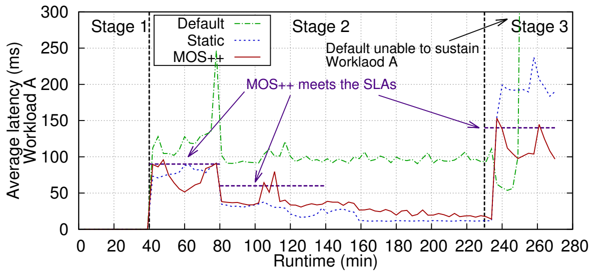
Adaptivity and Efficiency Evaluation. In our next experiment, we evaluate the adaptivity and resource efficiency of MOS++. We use a dynamically changing heterogeneous workload, which is issued concurrently by two tenants – `WorkloadA` and `WorkloadC`. The workload is generated by COSBench and is composed of three stages. In `Stage 1`, the workload is dominated by large objects (`WorkloadC`). At around 40 min, `Stage 2` begins with abrupt change in workload characteristics as the small-obj dominant workload (`WorkloadA`) instantaneously spikes and then gradually shifts down. This lasts for until around 200 min. Finally, at 230 min, in `Stage 3` there is another abrupt change as `WorkloadA` once again increases and dominates. With these three stages concatenated together we capture the behavior of MOS++ both under abrupt as well as gradual change in a dynamically changing workload in a multi-tenant environment. Figure 11(a) and Figure 11(b) plot the average throughput change and the average read latency change[4] of `WorkloadA`, respectively. Figure 11(c) plots the average bandwidth change of `WorkloadC`. Besides, Figure 12 depicts the breakdown of a variety of resources used in our tests.

As `Stage 1` begins, MOS++ achieves the same performance as `Static`. This is because MOS++ starts off with the same setup as `Static`. After around 5 min, MOS++ slightly outperforms `Static` by 12%, because MOS++ is SLA-aware and gradually reduces the network bandwidth originally allocated to `WorkloadA` and its performance reaches the highest after 5 min. Though `Static` achieves comparatively similar performance, it is not able to satisfy the SLA at 100% since its allotted network bandwidth is quickly saturated. In contrast, as observed in Figure 12, `WorkloadC` eventually uses up to 95% of all the available network resources (**R3** and **R5**

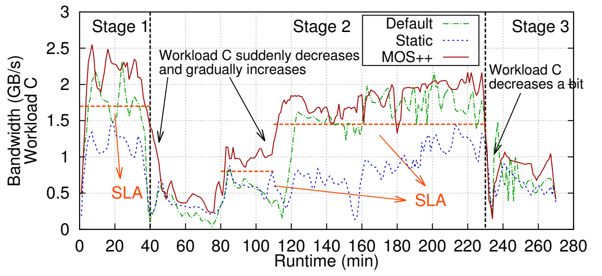
[4]We observed a similar trend for write latency.



(a) Throughput under `WorkloadA`.



(b) Average latency under `WorkloadA`.



(c) Bandwidth under `WorkloadC`.

Figure 11: Performance of MOS++ and the baselines under a dynamically changing heterogeneous workload. Purple dashed lines represent tenant-defined SLAs in terms of average latency (sla_{low}) for `WorkloadA`. Orange dashed lines represent tenant-defined SLAs in terms of bandwidth (sla_{low}) for `WorkloadC`.

in §3). Accordingly, MOS++ detects nothing significant in `WorkloadA` and decides to free up the CPU resource originally allocated to `WorkloadA` (recall that MOS++ and MOS starts with the same configuration). This clearly demonstrates that, rather than reserving resources statically, MOS++ elastically reprovisions the resources to achieve better resource efficiency while meeting the SLA.

`Stage 2` instantaneously begins at around 40 min and quickly spikes. At this time, MOS++ immediately start to react. Driven by the increasing `WorkloadA` and the specified SLA (90 ms average latency), MOS++ provisions proxies and object stores with more CPU cores. This can be observed in Figure 12 (**R2** and **R4**). Note in Figure 11(a), MOS++ improves the throughput of `WorkloadA` by up to 2.1× at around 60 min. As `WorkloadC`'s load decreases as `Stage 2` begins, MOS++ reduces the CPU cores allocated to `WorkloadC` and reassigns them to `WorkloadA`. Correspondingly, 58% of the network bandwidth allocated is reclaimed and marked as free and 21% is reallocated to `WorkloadA` (**R6**). `WorkloadA` spikes at around 60 min and gradually decreases while `WorkloadC` slowly increases. Accordingly, MOS++ adapts by reclaiming the extra CPU resources that are not needed for the current load/SLA for `WorkloadA`, and grabs back the network

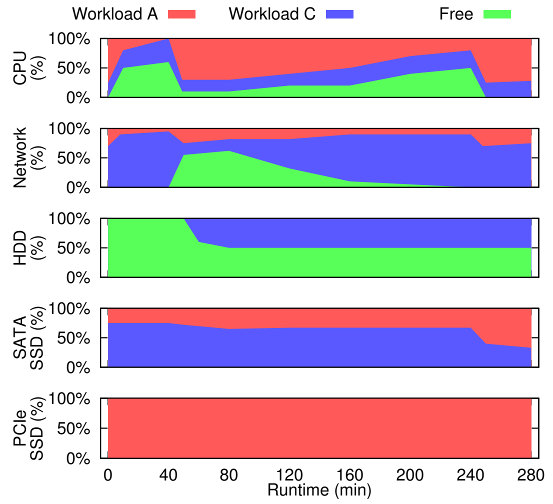


Figure 12: Resource allocation breakdown under dynamically changing heterogeneous workload.

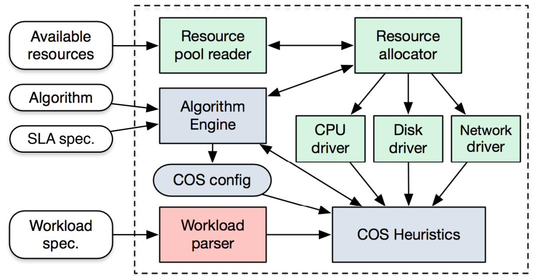


Figure 13: COSPerf architecture.

resource, which gets eventually saturated by `WorkloadC` (Figure 11(c)). Note that MOS++ allocates 50% of the HDD capacity to `WorkloadC` at 80 min, since the optimizer detects that a combination of HDD+SSD yields the most economical setup provided the given SLA (**R7**). As shown in Figure 11(a), MOS++ maintains the highest performance for both tenants while being resource efficient, whereas both `Default` and `Static` fail to do so, which further demonstrates the superiority of MOS++.

`Stage 3` begins at around 230 min with a sudden spike of `WorkloadA` and about half drop in `WorkloadC`. `Default` is unable to sustain `WorkloadA`, because the network bandwidth is mostly used up by `WorkloadC`. Again, MOS++ maintains the best performance for both tenants featuring the SLA-awareness and workload adaptivity.

8.2 Comparison of MOS and MOS++

In our next set of experiments, we use a simulation study to compare MOS and MOS++ for a large-scale setup.[5]

COSPerf Simulator. We design and implement a cloud object store simulator (COSPerf) based on the rules-of-thumb discussed in §3. Figure 13 depicts the architecture of COSPerf. A resource parser takes as input available resources, and sort them based on capability/capacity. The resources from these pools are then fetched by the resource allocator for launching resource drivers including a CPU driver, a disk driver, and a network driver. Resource allocator is driven by the algorithm engine, which is used to simulate the resource provisioning algorithm that is provided as a configurable pa-

[5]For comparison of MOS with `Static` and the monolithic `Default` setup please refer to our short paper [12].

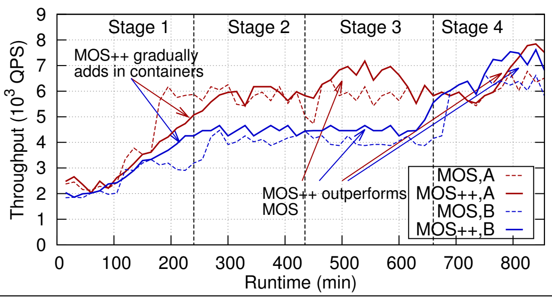


Figure 14: Throughput of MOS Vs. MOS++ under `WorkloadA` and `WorkloadB`.

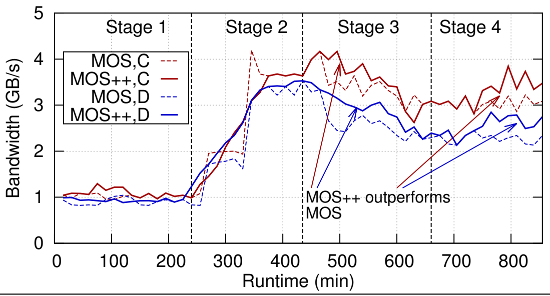


Figure 15: Bandwidth of MOS Vs. MOS++ under `WorkloadC` and `WorkloadD`.

rameter to COSPerf. The algorithm engine interacts with the resource allocator to keep track of the available resources and generates an object store configuration file. The algorithm engine also monitors the simulated performance of the launched microstores.

A cloud object store (COS) heuristics module takes workloads as input and predicts performance under a set of specified software/hardware configurations. The dynamic shifts in simulated performance due to online reconfiguration and workload changes are modeled and calculated by the COS heuristics module, which is built based on the extensive analysis and profiling of a real object store under various scenarios (§3). Thousands of experiments (each running for 15 minutes) were executed under varying load/configuration to further fine-tune our COS heuristics model.

Simulation Methodology. For simulation, we assume a pool of 50 server machines with diversified (heterogeneous) hardware configurations, including CPU, network, and storage devices. Specifically, we use the following pool of resources: (i) 3 32-core machines, 4 16-core, 31 8-core machines, and 12 4-core machines; (ii) 18 10 Gbps and 32 1 Gbps NICs; and (iii) HDD to SSD ratio was 70% to 30%. The workloads run for about 14 hours and we divide the workloads in four major stages. For analysis purposes, we focus on throughput in terms of QPS for small object workloads and bandwidth for large objects.

Simulation Evaluation. Figure 14 and Figure 15 show how MOS and MOS++ behave under dynamically-changing `WorkloadA-D`. The goal of our evaluation is to test under both abrupt and gradual change in a workload under multi-tenancy, which are emulated by our workloads. In `Stage 1`, as the load of small-object workloads (i.e., `WorkloadA` and `WorkloadB`) increases (as shown in Figure 14), MOS uses resources from the free resource pool to keep the resource utilization under control until the utilization stabilizes and falls back in the $[util_{low}, util_{high}]$ range. MOS++ also achieves the same goal but by allocating resources at a finer granularity. As a result, we do not see a sudden spike in QPS. Instead the transition is smoother, which allows both `WorkloadA` as well as `WorkloadB` to almost linearly increase QPS. In addition to maintaining resource utilization within the acceptable range, MOS++ also makes sure that SLA requirements are met.

As `Stage 2` begins, the load of large-object workloads (i.e., `WorkloadC` and `WorkloadD`) increases. As shown in figure 15, both MOS and MOS++ start adding resources to accommodate the increasing demand on network bandwidth. Again, MOS++ keeps fine-grained track of the resources at per-container basis and just allocates the right amount of resources to meet the tenant requirements while maximizing the resource efficiency. Starting around 310 min, however, MOS ends up adding more resources than needed to microstore C and D. Though MOS is able to lower down the network utilization from 95% to 85%, it causes a spike in performance.

From the beginning of `Stage 3` up until around 500 min, both MOS and MOS++ end up utilizing all the resources due to the increasing demands from all tenants. As shown in Figure 14 and Figure 15, MOS++ outperforms MOS by up to 25% for `WorkloadA`, and up to 31% for `WorkloadD`. This is due to the following design choices we make in MOS++: (i) MOS++ allocates resources at the container granularity; and (ii) MOS++'s optimizer generates a better resource provisioning plan that yields higher performance by exploiting all the available resources. After 500 min, the load decreases for `WorkloadC` and `WorkloadD`. As a result, MOS and MOS++ reclaim resources from microstore C and D to the pool of free resources. Finally, in `Stage 4`, the load further increases for `WorkloadA` and `WorkloadB`. As a result, MOS and MOS++ utilize the resources freed up in `Stage 3` from `WorkloadC` and `WorkloadD`. At this point, both MOS and MOS++ quickly detects performance improvement opportunity for `WorkloadB` as the throughput of `WorkloadB` is still at a low level, while more resources are also added into `WorkloadA` with the goal to maintain the CPU utilization within the "sweet" range. Hence, tenants will not see performance lost as the workload shifts. MOS++ further improves MOS's performance on `WorkloadA` and `WorkloadB` by up to 33% and 26%, respectively. This, again, demonstrates the superiority of MOS++ in effectively utilizing the limited resources for maximizing performance improvement and meeting tenants' SLAs.

Table 3 presents a summary of the average performance by combining all stages for `WorkloadA-D`. The results show that for small-object workloads, MOS++ achieves 12.4% better performance for `WorkloadA`, and 15.7% better performance for `WorkloadB`, compared to MOS. Similarly, average performance of MOS++ for large-object workloads is 15% higher for `WorkloadC` and 18.8% better for `WorkloadD` than that of MOS.

Workload	MOS	MOS++
A	4444 QPS	4994 QPS
B	3828 QPS	4429 QPS
C	2 GB/s	2.3 GB/s
D	1.6 GB/s	1.9 GB/s

Table 3: Average performance summary.

9. DISCUSSION

There are two limitations that are not fully addressed in our current implementation. First, although MOS supports multi-tenancy and heterogeneous workload separation, we limit the number of microstores to be launched based on workload characteristics (i.e., object sizes) to reduce the implementation complexity and reconfiguration overhead. Consequently, it limits the kinds of different workloads the system can effectively handle. Should a workload change its inherent characteristics, e.g., the object size distribution changes dramatically, and no longer fit well with any provisioned microstores, the system may end up doing reconfiguration thrashing. This in turn will lead to reduced performance. A possible solution is to perform online workload analysis and profiling at the load balancer/redirector side, and using the information to compute an optimal number of microstores and perform workload-to-microstore mapping on the fly. Such a dynamic detect-and-map system is part of our future work. Second, although MOS$_{++}$ is able to meet the SLAs by leveraging offline workload profiling and online optimization, it does not currently consider the profit, i.e., revenue, for the service provider and tenant utility, i.e., $perf/\$$, while provisioning the microstores. A feasible yet simple cloud-profit-aware solution can be to enhance our optimizer by incorporating the cloud pricing model and monetary profit. This aspect is orthogonal to our work, but can be easily incorporated into the design if needed.

10. CONCLUSION

In this paper, we have presented an experimental analysis of cloud object store, and proposed a set of rules-of-thumb based on the study. The rules provide practical guidelines for service administrators and online resource managers to better tune object store performance to application needs. The resulting system, MOS, outperforms extant object stores in multi-tenant environments. Furthermore, we build MOS$_{++}$ to enhance MOS by leveraging containers for fine-grained resource management and higher resource efficiency. Our experimentation reveals that it is possible to exploit the inherent heterogeneity within modern datacenters to better serve heterogeneous workloads across multiple tenants. Evaluation with our prototype implementation shows that MOS$_{++}$ improves performance by up to 89.6% and 79.8% compared to the default monolithic and statically configured object store setup, respectively. We have implemented COSPerf, a cloud object store simulator, to further verify the design choices of MOS$_{++}$. Results show that, by utilizing the same set of resources, MOS$_{++}$ achieves up to 18.8% performance improvement compared to the basic MOS.

Acknowledgments. We thank the anonymous reviewers for their comments. This work was sponsored in part by the NSF under Grants CNS-1405697 and CNS-1422788.

11. REFERENCES

[1] Arq. https://www.haystacksoftware.com/arq/.
[2] Ceph vs Swift. http://goo.gl/rtvrvg.
[3] Cloud backup with HP cloud. http://goo.gl/43bC5W.
[4] Control groups. https://goo.gl/KkCXAR.
[5] Docker container image for Swift object server. https://hub.docker.com/r/alivt/swift-object.
[6] Docker container image for Swift proxy sever. https://hub.docker.com/r/alivt/swift-proxy.
[7] Docker run preferences. https://goo.gl/SoF9Pc.
[8] IBM CPLEX optimizer. http://goo.gl/BA95mC.
[9] Linux traffic control. http://goo.gl/E5aQdq.
[10] Swfit storage policies. http://goo.gl/hRrySo.
[11] Volt git repo. https://github.com/VoltFramework/volt.
[12] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Taming the cloud object storage with mos. In *ACM PDSW*, 2015.
[13] A. Anwar, K. Krish, and A. R. Butt. On the use of microservers in supporting hadoop applications. In *IEEE CLUSTER*, 2014.
[14] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI*, 2010.
[15] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *ACM SOCC*, 2010.
[16] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. In *ACM SIGMOD*, 2012.
[17] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Cast: Tiering storage for data analytics in the cloud. In *ACM HPDC*, 2015.
[18] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Pricing games for hybrid object stores in the cloud: provider vs. tenant. In *USENIX HotCloud*, 2015.
[19] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In *ACM EuroSys*, 2013.
[20] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2009.
[21] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM ASPLOS*, 2013.
[22] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *IBM Research Report*, 2014.
[23] H. Greenberg, J. Bent, and G. Grider. Mdhim: A parallel key/value framework for hpc. In *USENIX HotStorage*, 2015.
[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, 2011.
[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.
[26] K. Krish, A. Anwar, and A. R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *IEEE CCGrid*, 2014.
[27] K. Krish, A. Anwar, and A. R. Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *IEEE MASCOTS*, 2014.
[28] G. Lee, B.-G. Chun, and R. H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *USENIX HotCloud*, 2011.
[29] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ACM ICAC*, 2010.
[30] J. Mars, L. Tang, and R. Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE CAL*, 2011.
[31] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg. Mantle: a programmable metadata load balancer for the ceph file system. In *ACM SC*, 2015.
[32] Y. Tanimura, S. Yanagita, and T. Hamanishi. A high performance, qos-enabled, s3-based object store. In *IEEE CCGrid*, 2014.
[33] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *USENIX FAST*, 2011.
[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI*, 2006.
[35] Q. Zheng, H. Chen, Y. Wang, J. Duan, and Z. Huang. Cosbench: A benchmark tool for cloud object storage services. In *IEEE CLOUD*, 2012.