Taming the Cloud Object Storage with MOS

Ali Anwar*, Yue Cheng*, Aayush Gupta†, Ali R. Butt*
*Virginia Tech, †IBM Research – Almaden
{ali,yuec,butta}@cs.vt.edu, {guptaaa}@us.ibm.com

Abstract

Cloud object stores today are deployed using a single set of configuration parameters for all different types of applications. This homogeneous setup results in all applications experiencing the same service level (e.g., data transfer throughput, etc.). However, the vast variety of applications expose extremely different latency and throughput requirements. To this end, we propose MOS, a Micro Object Storage architecture with independently configured microstores each tuned dynamically for a particular type of workload. We then expose these microstores to the tenant who can then choose to place their data in the appropriate microstore according the latency and throughput requirements of their workloads. Our evaluation shows that compared with default setup, MOS can improve the performance up to 200% for small objects and 28% for large objects while providing opportunity of tradeoff between two.

1. Introduction

Cloud object stores, such as S3 [1], Google Cloud Store (GCS) [6], Swift [7] and Ceph [3], have become the most widely used form of cloud storage in recent years. They 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, with HTTP-based RESTful APIs for data management. These desirable features coupled with the advances of virtualization infrastructure are driving the uses of cloud object stores by a myriad of applications, ranging from web applications [12] to backup services [4, 5] , and big data analytics frameworks [8, 18].

Cloud object stores today are deployed using a single set of configuration parameters for all different types of applications. This homogeneous setup results in all applications experiencing the same service level (e.g., average latency per request, data transfer throughput, and queries per second (QPS)). However, the vast variety of applications expose extremely different latency and throughput requirements. For example, a social networking or photo sharing application requires low latency to keep a responsive user experience, whereas backup services can tolerate higher latency but require sustained high throughput. Extant object storage services compromise application performance to gain the flexibility advantages.

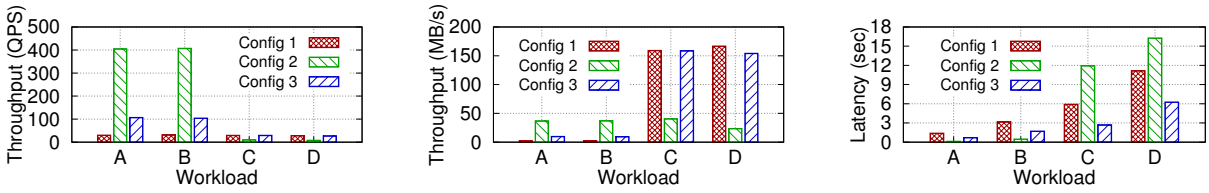
From the cloud provider's perspective, supporting the widely different workloads of different applications using a single homogeneous configuration results in a loss of optimization opportunities. For example, a photo sharing application such as Instagram [10], would have a large number of small-medium sized files (e.g, KB- to MB-level image objects) In contrast, an enterprise backup application such as Arq [2] or Tarsnap [9], consists largely of write requests for large cold archive files with reads only sparsely arising.

The situation is further complicated by the fact that due to regular system upgrades, data centers hosting object stores are becoming increasingly heterogeneous. However, with the "one-size-fits-all" style of object store deployment, it is impossible to match each set of specific types of hardwares 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 and weaker CPU power. Under this scenario, meeting service level agreement (SLA) requirement for one of the workload is challenging, and may require, (i) adding hardware resources that may not be a ideal fit for other workloads, (ii) software tuning that may affect the performance of the other workloads.

In this paper, we argue that *it is more beneficial to separately entertain these workloads in finer-grained object stores launched on sub-clusters formed using the available hardware resources*. To demonstrate this, we studied different types of workloads that are commonly used in practical scenarios. we looked at four different real-world applications that use cloud object storage as listed in Table 1. We deployed and evaluated Swift [7] in a multi-tenant environment

PDSW2015, November 15-20, 2015, Austin, TX, USA.

http://dx.doi.org/10.1145/2834976.2834980



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

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

Workload	Workload characteristics		Application scenario
	Object size	Distribution	
`Workload A`	1-128 KB	G: 90%, P: 5%, D:5%	Web hosting
`Workload B`	1-128 KB	G: 5%, P: 90%, D:5%	Online game hosting
`Workload C`	1-128 MB	G: 90%, P: 5%, D:5%	Online video sharing
`Workload D`	1-128 MB	G: 5%, P: 90%, D:5%	Enterprise backup

Table 1: Different types of workloads and application scenarios used for testing the behaviors of object stores. G means GET, P means PUT and D means DELETE.

using COSBench [22] as workload generator configured for the four types of workloads mentioned previously. Swift is a popular object store implementation provided by OpenStack that is increasingly becoming de facto cloud computing software platform. In motivational tests, we used three different Swift configurations (setups). We ran COSBench workload generator on designated machines to saturate the Swift store. Each benchmark was executed for 15 minutes with ramp up time of 2 minutes. We used two nodes as proxy servers in each of the configuration. To simulate the datacenter heterogeneity, one of the proxy server was a 32-core machine and the other was 8-core. The proxy server running on a 32-core machine was connected to the storage nodes via 1 Gbps interconnect while the proxy server on an 8-core machine was connected via 10 Gbps network. In addition, four 32-core machines were used as storage nodes. Each storage node had 3 SATA SSDs attached as a storage device. The storage nodes were configured in such a way that they do not act as performance bottleneck for any of the studied configuration:

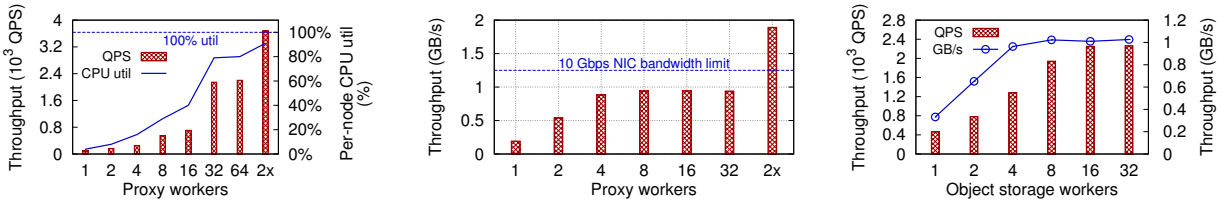
Config. 1: The default monolithic Swift setup. Both 8-core and 32-core machines acted as proxy server. It was handled by all resources and round robin DNS was used for load balancing. **Config. 2:** All resources were divided into two sub-object stores, one for small objects workload and the other for large objects. One 8 core machine (connected via 10 Gbps) serves as proxy for Workload A and B, and Workload C and D uses one 32-core machine connected via 1 Gbps NIC. **Config. 3:** One 32-core machine (connected via 1 Gbps) served as proxy for Workload A and B while one 8-core machine (connected via 10 Gbps) served as proxy for Workload C and D.

Figure 1 shows the comparison of performance achieved by different configurations. As shown in Figure 1(a), separating proxy servers for different workloads improved the overall QPS by 700% for Configuration 2 and by 225% for Configuration 3 as compared with the default Swift setup. It is interesting to note that even though the Configuration 2 resulted in very high QPS for small objects (Workload A and B), 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 (Workload C and D). On the other hand, in Configuration 3 the throughput for large objects remained same. Similarly, the latency of Configuration 3 is also less than that achieved by the default configuration for all the workloads (Figure 1(c)). Configuration 2 provides best and worst latency for small and large object workloads, respectively. We also observed that switching NIC's of proxy servers in configuration one resulted in similar results.These results demonstrate that comprehensive study of the impact of different configuration on performance is needed.

Key Insights 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 experience a dramatic performance down. This is because the available network bandwidth is exhausted to transfer TCP packets containing payload for large object 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 in Configuration 2 and 3 gives fair chance to small object workloads to be properly handled by the storage nodes. Thus, there is an urgent need for cloud object stores to efficiently utilize the available resources to make sure that each tenant is treated as a first class citizen.

In this paper, we make the following contributions:

1. We evaluate the impact of conventional object storage configuration on performance and resource efficiency by conducting experiments on a local Swift testbed.
2. 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.
3. Based on our analysis, we design MOS, an object store that (1) dynamically provisions fine-grained microstores, each configured with different combination of hardware



(a) Proxy as a knob for small-object workload. (b) Proxy as a knob for large-object workload. (c) Object storage node as a knob.

Figure 2: Different software/hardware configuration options. In (c), small-object workloads refer to bars (QPS) while large-object ones linepoints (GB/s).

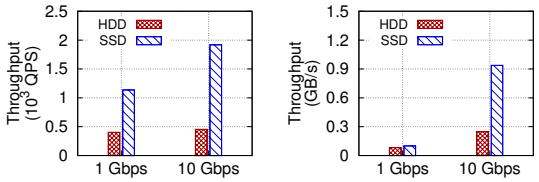
and software options, and (2) exposes the interfaces of microstores to the tenants according to their application requirements.

4. We demonstrate using simulation that MOS results in up to 2× more QPS compared to the baseline object store setup, as well as good resource utilization using a fixed set of resources.

2. A Case for Cloud Object Storage

In this section we present a detailed analysis of how object stores behaves under various software and hardware configuration options. To this end, we classify the workloads on the basis of object size. For workloads dominated by small objects (at KB level) the interested metric is the QPS and response latency while for workloads dominated by large objects (at MB-GB level) throughput in terms of MB/s or GB/s is the most important metric. We then study the effects of tuning various configuration *knobs* and develop a set of useful configuration *rules-of-thumbs* that can be used to guide the design of MOS.

Proxy as a knob First, we study the effect of scaling proxy nodes on workload performance. 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). We vary the computational capacity of proxy by increasing proxy's allotted CPU cores. Figure 2(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 closed to 85% (bounding the throughput) with both 32 and 64 proxy workers, implying that CPU becomes the bottleneck. 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. We perform the same test with large object workloads. In Figure 2(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 proxy node. This is because for large object intensive workload, the performance is constrained by the network bandwidth before CPU is saturated. We make the following observations – proxy's computational capacity can be the bottleneck for workloads dominated with small objects, whereas the network bandwidth is the most precious resource for large-object intensive workloads.



(a) Small-object workloads. (b) Large-object workloads.

Figure 3: Hardware effect ⟨storage devices, NICs⟩ on throughput.

Storage as a knob Second, we study the effect of scaling object storage nodes on workload performance. As shown in Figure 2(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 strengths our observation that CPU is the most important resource for small-object workloads. For large-object workloads, in contrast, the network constraint is quickly reach with only 4 object storage workers. This is because for large objects the performance is bottlenecked by the network given higher disk bandwidth (recall that each storage node has 3 SATA SSDs).

Hardware as a knob Third, we study the effect of varying storage device and network connectivity on workload throughput. Figure 3(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. The interesting observation shows that small-object intensive workload is more sensitive to the storage devices rather than the network bandwidth, implying that in some situation such kind of workload, which does not impose extremely high requirement, may be efficiently handled using weaker network interconnect but good 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 3(b)), which clearly

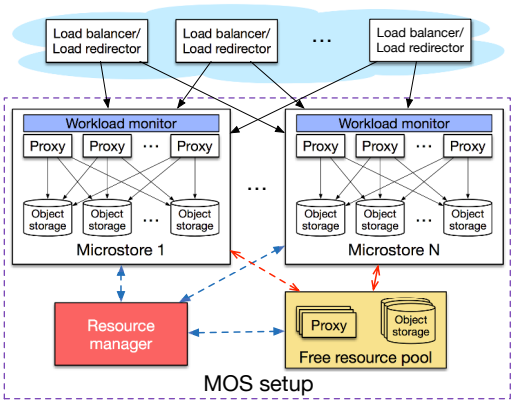


Figure 4: Overview of MOS design.

indicates such kind of workloads prefer good network interconnect as first-priority resource.

3. MOS Design

We design MOS in light of the *rules-of-thumbs* developed in §2 and existing workload and datacenter heterogeneity. Conventional cloud object storage, such as OpenStack Swift, adopts a monolithic storage architecture, where all tenants/workloads share the storage resources. This design is simple in implementation and configuration but is not necessarily resource efficient. For example, a backup workload consisting mostly of large objects may easily consume all the available network bandwidth that might have been assigned to workloads dominated with small objects, thus wasting the CPU resources that would have been allocated to serve the small object requests.

To address this issue, 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 depicted in Figure 4, MOS consists of two layers: (1) **Microstores:** consists of multiple instances of object stores, each called a *microstore* that is allocated a subset of proxy nodes and storage nodes that matches the requirements of the workload it is meant to support. The number of microstores configured in a deployment of MOS depends on the kinds of workloads that need to be supported. (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. Resource manager makes decisions about when and how to add or redistribute resources. The decisions are made using a simple greedy algorithm detailed in Algorithm 1.

Algorithm 1 takes as input $microstores$, a vector of all microstores storing statistics (including hardware configuration, current load served, and the resource utilization such as CPU and network bandwidth utilization) of all microstores. 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

Algorithm 1: 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

```
1  begin
2      microstores.hw ← init(free_pool)
3      while true do
4          foreach ms in microstores do
                       // periodically collect monitoring stats
5              if util_low ≤ util(ms.hw) ≤ util_high then
6                  ms.firstTime ← true
7                  continue
8              else
9                  if ms.firstTime then
10                     ms.firstTime ← false
11                     ms.toChange ← 1
12                 else
13                     ms.toChange ← ms.toChange * 2
14                 if util(ms.hw) > util_high then
                                   // to add in more resources
15                     ms.hw ← ms.hw + ms.toChange
                       /* allocate resource from free resource pool   */
16                     alloc(free_pool, ms, ms.toChange)
17                 else if util(ms.hw) < util_low then
                                       // to remove resources
18                     ms.hw ← ms.hw − 1
                       /* return resource to free pool               */
19                     dealloc(free_pool, ms, 1)
20         sleep(epoch)
```

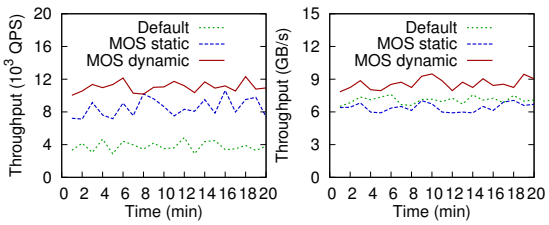
polls each microstore. In each iteration, if the resource utilization (fetched using $\texttt{util}(ms.hw)$) of one microstore lies within a pre-defined threshold range, the algorithm simply iterates to the next microstore. If the microstore is in sub-optimal state, the algorithm decides to *quadratically* add or *linearly* remove resources. This way it is guaranteed that it will not overdo when deallocating resources while being able to quickly respond to sudden workload increases.[1]

4. Preliminary Results

In this section, we present our simulation based preliminary results to demonstrate how MOS performs in practice. The simulator was designed based on observations discussed in §2. The shifts in simulated performance due to configuration changes was calculated based on the trends learned through extensive analysis of how object store behaves under various scenarios. More than one thousand experiments (each ran for 15 minutes) were executed under varying load/hardware configuration to further fine-tune the simulator. We use a pool of 50 machines with heterogeneous hardware configurations (e.g., CPU, network, and storage devices, etc.): 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) an HDD to SSD ratio of $70\% : 30\%$.

We first conduct simulation under constant load to compare the performance of MOS with the default single object store instance setup. We then simulate a dynamically-changing long workload that exhibits heterogeneous characteristics to test the effectiveness of MOS's online resource

[1] As part of our future work, we are exploring more sophisticated techniques for handling adversary workload shifts.



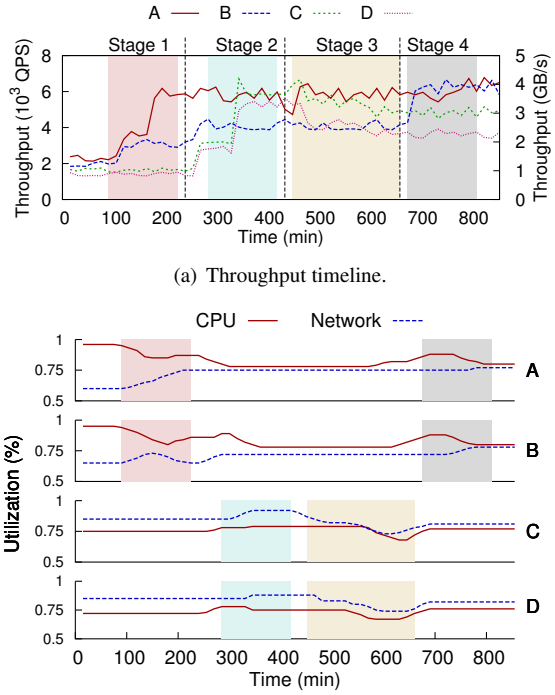
(a) Comparison of the QPS. (b) Comparison of GB/s.

Figure 5: Aggregated throughput seen by different setups.

provisioning algorithm. Both simulations run the mix of the four workloads listed in Table 1.

In static case, all 32-core and 16-core along with 3 8-core machines were used as proxy servers. Each proxy server was connected via 10 Gbps network. Rest of the machines were used as object storage nodes. We test two baseline cases – the default object storage setup (`Default`), and a static MOS setup (`MOS static`) where resources are divided according to the rules from §2, on the same set of proxy servers. We further run dynamic MOS (`MOS dynamic`) on the same set of resources to show the benefit of Algorithm 1. Figure 5 shows the results. As observed, at any given time the QPS improvement of `MOS static` against `Default` is greater than 180% whereas the maximum decrease in throughput to be less than 8%. This is because `MOS static` isolates and preserves resources so that Workload C and D will not eat up the whole network bandwidth. The aggregated GB/s is slightly reduced as the network bandwidth utilization is almost 95% (bounding the performance of `MOS static`). `MOS dynamic` is able to improve both the QPS and GB/s since the online provisioning algorithm effectively detects object storage under-utilization and eventually move the under-utilized resources to serve as proxies for Workload C and D. As a result, the average network utilization of microstore C and D is reduced to 85% while the overall performance improvement is seen.

Figure 6 shows how MOS behaves under dynamically-changing Workload A-D. The test runs for about 14 hr and we divide the workloads in four major stages. In Stage 1, the load for small objects, i.e., Workload A and B, increases. As the load increases (in Figure 6(a)), MOS uses resources from free resource pool to keep the resource utilization under control untill the utilization is stabilized and lies in $[util_{low}, util_{high}]$ (in `A` and `B` of Figure 6(b)). This results in increased QPS for Workload A and B and sustained throughput for Workload C and D. In Stage 2, the load for large objects (i.e., Workload C and D) is increased untill it reaches the same level as in static simulation of Figure 5. Starting around 310 min, MOS adds more resources in microstore C and D, hence lowering down the network utilization from 95% to 85% (`C` and `D` of Figure 6(b)). In Stage 3, the load of Workload C and D is decreased. As a result, MOS reclaims resources from microstore C and D to the pool, as seen in gradual increase in network utilization. Finally, in Stage 4, the load for Workload A and B is further increased and MOS



(a) Throughput timeline.

(b) Resource utilization timeline.

Figure 6: Throughput and resource utilization timeline under dynamically-changing workload. The overall workload is a combination of Workload A, B, C, and D.

grabs resources freed up in Stage 3 from Workload C and D. We see that the throughput for Workload B is improved by 49% and that for Workload A is kept the same. MOS quickly detects performance improvement opportunity for Workload B as the throughput of Workload B is still at a low level, while more resources are added into microstore A with the goal to maintain the CPU utilization within the "sweet" range (hence, tenants will not experience performance lost as the workload shifts). This demonstrates that as workloads shift, MOS is able to effectively and quickly tune the configurations of microstores to provide high performance guarantee while maintaining high resource utilization.

5. Related Work

hatS [19] and Skute [13] proposes a fault-tolerant and scalable replication scheme for HDFS and cloud storage, respectively. MOS has a wider scope – as a cloud object store, it provides fault tolerance functionality; more importantly, it aims to improve the overall performance by efficiently exploiting the data center and workload heterogeneity. [11] proposes a metric based on CPU, I/O wait and memory usage that are critical for Hadoop. Similarly, MET [17] proposes several system metrics that are critical for a NoSQL database and highly impacts server utilization's estimation. In contrast, we focus on automated elasticity for cloud object store. ϕSched [20] and Walnut [14] propose sharing of hardware resources across hitherto siloed clouds of different types. CAST [15] and its extension [16] perform coarse-

grained cloud storage (including object stores) management for data analytics workloads. At a finer granularity, MOS explicitly partitions the conventional single storage setup into multiple dynamically fine-tuned microstores, each serving a particular type of workload. Finally, IOFlow [21] solves a similar problem by providing a queue and control functionality at two OS stages – the storage drivers in the hypervisor and the storage server. Unlike IOFlow, MOS requires no OS level changes.

6. Conclusion

In this paper, we first presented results of our exhaustive study of cloud object store. Second, we proposed a set of rules to help cloud object store administrator to efficiently utilize resources. Third, we presented MOS which can outperform extant object store under multi-tenant environment. Our analysis shows that it is possible to exploit heterogeneity inherited by modern datacenter to the advantage of object store providers. Simulation results show that MOS outperforms extant object store under multi-tenant.

Acknowledgments This work was sponsored in part by the NSF under CNS-1405697 and CNS-1422788 grants.

References

[1] Amazon s3. http://aws.amazon.com/s3/.

[2] Arq. https://www.haystacksoftware.com/arq/.

[3] Ceph. http://ceph.com/.

[4] Cloud backup with hp public cloud. http://www.hpcloud.com/solutions/backup.

[5] Emc cloud object storage. http://www.emc.com/storage/atmos/atmos.htm.

[6] Google cloud storage. https://cloud.google.com/storage/.

[7] Openstack swift. http://docs.openstack.org/developer/swift/.

[8] Ozone: An object store in hdfs. http://hortonworks.com/blog/ozone-object-store-hdfs/.

[9] Tarsnap - Infrastructure. http://www.tarsnap.com/infrastructure.html.

[10] What Powers Instagram: Hundreds of Instances, Dozens of Technologies. http://instagram-engineering.tumblr.com/post/13649370142/what-powers-instagram-hundreds-of-instances.

[11] A. Anwar, K. Krish, and A. R. Butt. On the use of microservers in supporting hadoop applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 66–74. IEEE, 2014.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[13] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 205–216. ACM, 2010.

[14] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM, 2012.

[15] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 45–56, New York, NY, USA, 2015. ACM.

[16] Y. Cheng, M. S. Iqbal, A. Gupta, A. R. Butt, and V. Tech. Pricing games for hybrid object stores in the cloud: provider vs. tenant. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, pages 20–20. USENIX Association, 2015.

[17] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2013.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[19] K. Krish, A. Anwar, and A. R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 502–511. IEEE, 2014.

[20] K. Krish, A. Anwar, and A. R. Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 255–264. IEEE, 2014.

[21] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[22] Q. Zheng, H. Chen, Y. Wang, J. Duan, and Z. Huang. Cosbench: A benchmark tool for cloud object storage services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 998–999. IEEE, 2012.