COSBench: Cloud Object Storage Benchmark

Qing Zheng*, Haopeng Chen
School of Software, Shanghai Jiao Tong University, Shanghai, China
{ mark, chen-hp } @ sjtu.edu.cn

Yaguang Wang, Jian Zhang, Jiangang Duan
Intel Asia-Pacific R&D Ltd., Shanghai, China
{ yaguang.wang, jian.zhang, jiangang.duan } @ intel.com

ABSTRACT

With object storage systems being increasingly recognized as a preferred way to expose one's storage infrastructure to the web, the past few years have witnessed an explosion in the acceptance of these systems. Unfortunately, the proliferation of available solutions and the complexity of each individual one, coupled with a lack of dedicated workload, makes it very challenging for one to evaluate and tune the performance of different systems. To help address this problem, we present the *Cloud Object Storage Benchmark* (COSBench). It is a benchmark tool that we have developed at Intel with the goal of facilitating both performance comparison and system optimization of these systems. In this paper, we describe the design and implementation of this tool, focusing on its extensibility and scalability. In addition, we discuss how people can use this tool to perform system characterization and how the latter can facilitate system comparison and optimization. To demonstrate the value of our tool, we report the results of our experiments conducted on two Swift setups we built in our lab. We also share some of our experiences in turning our setups to achieve higher performance.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation

Keywords

Object Storage, Benchmark Tool

1. INTRODUCTION

With our world being increasingly connected and enriched by newly emerged computing technologies, the data produced is gigantic [31]. To combat such data tsunami, object storage has been introduced to fulfill the common need of a scale-out storage infrastructure, with acceptance of these systems growing every year [1]. However, the proliferation of object storage systems and the complexity of each individual one makes it very difficult for one to choose the best system. In order to better understand existing options, careful performance characterization of each system is typically needed, so one can quantitatively compare different candidates and then pick out a system that can best accommodate its specific requirements. For example, people dealing with large objects may confidently steer their attention away from a system that only excels in handling small objects.

Tuning systems to achieve maximum performance is also challenging. It entails not only choosing the optimal values in a variety of software configurations, but also assigning the correct hardware to different components constituting the system. In order to approach the best combination, numerous experiments have to be conducted across the large spectrums of possibilities at different dimensions, which can be quite time-consuming.

To alleviate these problems, we present the *Cloud Object Storage Benchmark* (COSBench). It is a benchmark tool that we have developed to help people better compare different hardware and software stacks as well as better identify bottlenecks to drive system tuning and optimization. Capable of generating workloads that mimic diverse applications, this tool allows people to characterize systems under their own usage patterns. In addition, a lot of attention has been paid to ensuring the tool's scalability and extensibility, so that people can easily simulate a large number of concurrent clients and adapts these clients to a myriad of storage systems. In order also to secure ease-of-use, we have designed our tool to be operated with either web browsers or command-line utilities, making it friendly not only to human users, but automation tools as well.

Although file systems and block storage can also be employed to manage data, we focus on object storage systems with the following reasons. Firstly, there are many existing tools [26, 17, 25, 16] available for measuring other types of storage systems. Besides, recent years have seen a trend in building file systems and relational databases upon object storage infrastructures [19, 27, 28, 29], which compels our attention to the performance of these underlying services. Finally, the lack of a nontrivial workload for novel object storage systems best warrants our development of the tool.

In this paper, we describe the design and implementation of the COSBench, with discussions on how people can use it to perform system characterization and how the latter can facilitate system comparison and optimization. We

*This paper was performed in Intel where the author is an intern.

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

Figure 1: Object storage interface

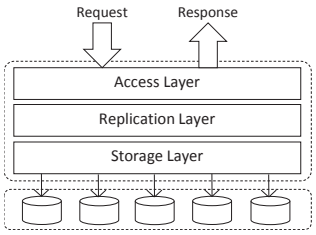


Figure 2: Architecture of object storage systems

also share the results of our experiments conducted on two different Swift setups we deployed in our lab.

The rest of this paper is organized as follows. Section 2 gives an overview of object storage systems while Section 3 discusses system characterization. Section 4 introduces the workload model used in the COSBench with details of the tool itself covered in Section 5. Section 6 and 7 share the results and the experiences we have obtained from our experiments, followed by Section 8 which reports the related works. The paper is concluded in Section 9 where future works are also described.

2. OBJECT STORAGE OVERVIEW

In this section we present an overview of the object storage by discussing the common motivation, storage interface, and system architecture shared among different implementations.

2.1 Motivation

Object storage is largely motivated by the gap between file systems and block storage. On one hand, many applications nowadays find themselves less dependent on traditional POSIX file systems. For instance, applications dealing with images can sit comfortably upon a file system that only supports a flat namespace and weak consistency. Content sharing applications can also painlessly devise their own ways to handle concurrent modifications without resorting to file system locks. On the other hand, although a reduction of file system features frequently fuels scalability, developers still need storage services to free them from the burden of managing blocks and various other storage metadata. To this end, object storage systems are often implemented with eclectic designs: they are less sophisticated than file systems but still more intelligent than block devices.

The rise of cloud computing has also assisted the emergence of object storage. Object storage echoes the very spirit of cloud computing. It provides people with an unlimited and on-demand data depository accessible from anywhere in the world, and it helps to achieve data consolidation and economy of scale, both facilitating cost-effectiveness. With cloud computing getting increasingly adopted, so are object storage systems.

2.2 Storage Interface

A careful design of the storage interface is essential for object storage to become a building block in web computing: (i) the interface should ensure ubiquitous access on the part of millions of applications running all over the Internet; (ii) appropriate security mechanisms have to be properly incorporated into the interface to facilitate user authentication and data encryption; (iii) it is also necessary to keep the interface stateless as session management notoriously hinders the scalability of systems; (iv) object storage must eschew binding its interface to any specific programming languages in favor of a more interoperable approach. With all above criteria honored, object storage systems are designed as RESTful web services and are accessed via HTTP requests, as is exemplified in Figure 1.

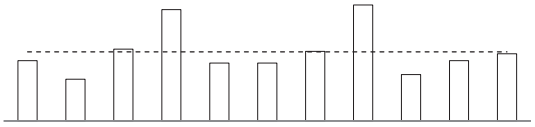
To use object storage systems, people create *containers* and put *objects* into these containers for storage. Objects are just like files in regular file systems, though they cannot be locked or updated partially. Containers are identical to directories except that they cannot be nested. Both objects and containers are identified by unique names distinguishing one from each other. Notwithstanding the resemblance among the interfaces of various object storage systems, details do differ. While efforts [3] have been made towards a standard protocol, time is yet needed before an extensive acceptance is eventually reached.

2.3 Architecture

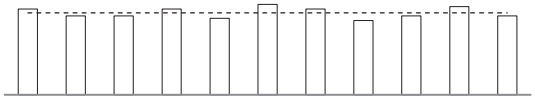
The techniques related with implementing object storage systems can be stratified into three distinct layers: storage layer, replication layer, and access layer, as illustrated in Figure 2. The storage layer is responsible for storing objects to physical devices. The key issue the storage layer has to deal with is performance: it has its own discretion in optimizing either for the read performance or the write. As most storage layers use file systems to handle their workloads, it is then important to choose the correct file system and to mount it with the proper options. A benchmark tool can help people compare different file systems and options, and tune the storage layer to its best capacity.

The replication layer sitting upon the storage layer holds all the secrets of transforming a pool of storage devices to a web-scale data serving platform. As a first step, the replication layer should keep an eye on the durability of each object stored in the system. After all, hardware faults inevitably occur and high quality devices often incur prohibitive expenses. To this regard, some systems create multiple copies to offset disk failures, others employ erasure codes [24] to break each object into redundant fragments. However, in both cases, it follows that the replication layer should start to take care of the consistency of the copies or fragments associated with each object. This is often resolved by rebuilding missing copies and quarantining corrupted ones. One key issue here is to shorten the inconsistency window as much as possible while also averting impact on the main activities of the system. With a benchmark tool in hand, people can better evaluate and study these impacts, obtaining insights to guide system optimization.

Flexibility and scalability are also addressed in the replication layer, with this layer bearing the responsibility of scattering the incoming workload to all storage servers. Load



(a) a system exhibiting poor predictability but lower average response time.



(b) a system exhibiting great predictability though higher average response time.

Figure 3: Characterizing the performance

balance in fact also encompasses assigning part of the workload to newly joined servers and revoking from overloaded or faulty ones. More judicious algorithms may also take hot spots into consideration by adaptively rebalancing the workload accordingly. However, the key issue here is to keep the system efficient, available, and manageable. To achieve these targets, a benchmark tool can be employed to drive and verify better system designs.

Finally, the access layer on top of the system serves as the interface to external applications. One crucial duty of the access layer is to deal with all kinds of failures, thus to ensure high availability. In most cases, it is important to keep a system both writable and readable in spite of the presence of errors. The access layer is also regarded as the last resort for achieving high performance. For example, some people may employ concurrent reads to reduce latency at the expense of throughput; others may introduce asynchronization to boost write performance with both durability and consistency compromised. One important requirement of the access layer is to delicately prioritize and balance different system properties, with the goal of best satisfying the need of their applications. To this regard, a benchmark tool capable of generating customizable workloads will be extremely valuable to people exploring these tradeoffs.

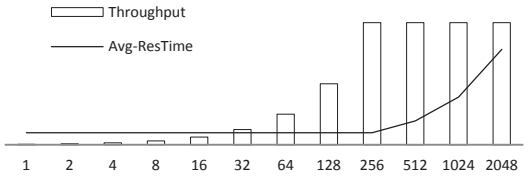
3. SYSTEM CHARACTERIZATION

In this paper, we focus on system characterization. As will be discussed in this section, data obtained from characterization can be quite useful in many possible ways: (i) it provides tangible insights for one to compare different object storage offerings; (ii) it brings knowledge and experience on better system deployment that are important for tuning performance; (iii) it establishes a baseline performance by which further research or engineering efforts can be envisaged and channelled; and finally, (iv) the profiling data acquired during system characterization can help inspire new ideas of improvements and drive optimization.

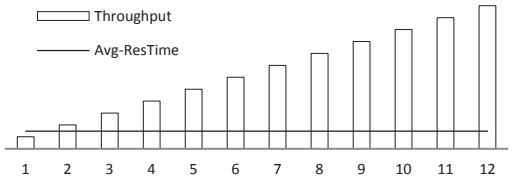
In this section, we introduce three different aspects that one should bear in mind during system characterization.

3.1 Performance

The first aspect people need to focus on is undoubtedly the performance. One interesting issue regarding performance is its involvement of two related but competing properties



(a) an ideal system that keeps scale until its capacity is reached. The horizontal line represents numbers of clients.



(b) an ideal system that scales in proportion to its hardware resources. The horizontal line represents resource units.

Figure 4: Characterizing the scalability

displayed by a system: (i) the efficiency of a system which is expressed by throughput or average response time, and (ii) the predictability of a system which is associated with the distribution of its response times. To achieve both is difficult; one has to meticulously design the system to reconcile the inherent conflict between the above two properties. Since most SLAs stipulate bounds for latencies, it is then important to ensure the predictability of a system in order to conform to these agreements. Ideals of such enforcement often turn to amortization: the time spent on certain unforeseeable and high-latency events is split among all requests, thus diminishing the variance of their response times. Unfortunately, the ensuing extra efforts taken by the system to perform such actions inevitably dilute resource utilization, contracting system efficiency with an increased average response time and a decreased system throughput.

Examples of such efforts include the concurrent reads and the asynchronous writes we mentioned in Section 2, which are also illustrated in Figure 3-a and 3-b. Note that the key point behind these tradeoffs is that we must take both efficiency and predictability into consideration when performing system characterization. Therefore, a legitimate benchmark tool should be able to measure both properties, facilitating unbiased analyses against these systems. A system with better performance is one that exhibits higher efficiency while maintaining sufficient predictability. While results of such an analysis can help people better evaluate and compare different object storage systems, they are equally helpful in forming baseline performance for future optimization. The core ideal here is to decide on an acceptable latency, and then to maximize the throughput as much as possible.

3.2 Scalability

Different from the previous aspect which focuses on the balance between efficiency and predictability, scalability seeks to explore the performance under a dynamic context marked by either an increasing number of clients or hardware resources, as is shown in Figure 4-a and 4-b. In the first case,

the hardware is fixed but the number of clients is raised exponentially. While an ideal system can answer this situation with a constant average response time and an increasing throughput proportionate to the growing number of clients, real systems are often subject to the contentions of various resources such as CPU, disks, networks, or software locks, thus cannot exhibit such perfect trends. The core ideal here is to improve the parallelism of a system to sustain higher performance on a given hardware configuration.

Unfortunately, every system has its maximum capacity: once it is saturated, more clients will have to compete with each other for resources, resulting in unacceptable response times; the best possible case is shown in Figure 4-a.

Ideally, after a system is saturated, in order to sustain more clients, only commensurate amount of hardware will have to be provisioned to stabilize the response time and to keep the throughput increasing accordingly, as is shown in Figure 4-b. However, in real situations, adding hardware resources does not necessarily earn proportionate performance enhancement, if at all. Many factors can be held culpable for impeding such improvements: metadata updating, load imbalance, or hardware limits. The core ideal here is to identify and eliminate these bottlenecks so as to better harness the underlying hardware resources, allowing the system to grow into thousands of nodes or even beyond.

By projecting performance to the dimensions of clients and resources, scalability plays a significant role in comparing different object storage systems. A better system is one that can engage more clients with fewer servers. Besides, results of scalability tests can also be studied to uncover system bottlenecks, which provides insights to guide system tuning. Note that it is crucial for a benchmark tool to be scalable as well, so that a large number of clients can be simulated to stress these storage systems in the first place.

3.3 Hardware Profiles

Finally, our last aspect of system characterization lies in the status of various hardware resources including CPU, memory, disks, and networks. To start with, one can always use these data to identify pathological resource utilizations and then take corresponding actions to get rid of them. For example, high CPU utilization is often associated with inadequate CPU power and may be resolved by either switching to more powerful nodes, or provisioning more nodes to share workloads. Profiling data can also be used to examine the balance among different servers or different CPU cores or disks within a single server. Any blatant imbalance exposed here is either a potential path for future optimization, or a symbol suggesting hardware faults. For example, it is not uncommon for a small group of disks to hold the "hot spot" data and hence to be more frequently accessed. To tackle this, an extra layer of cache could be deployed to level off the exorbitant burden of these disks, thus restoring balance. Moreover, profiling data can assist people to better invest money on hardware resources by revealing the resource usage patterns of a given system in the face of a certain workload. For example, if it is the CPU power that matters to the overall performance, it might not be that sensible for one to still spent money on high-end disks.

4. WORKLOAD MODEL

In this section, we present the workload model that underlies the COSBench system. A workload model abstracts the common patterns shared among a variety of object storage applications; it is crucial to make the model flexible and representative so that diverse usage patterns can be readily simulated to yield informative observations.

Table 1: Storage Interface

Operation	Description
`LOGIN`	retrieve a token representing an identity
`READ`	download an object
`WRITE`	upload a new object
`REMOVE`	delete an existing object
`INIT`	create a new container
`DISPOSE`	delete an empty container

4.1 Storage Interface

Storage interfaces are protocols defined by object storage systems to expose service. As there is no widespread standard available at this moment, different systems assume different protocols. In order for our workload model to work with most object storage systems, we have defined our storage interface as a small bundle of common operations seen in various specific protocols. So far, we have six core operations defined in our interface, as summarized in Table 1. These operations are sufficient for one to take on tasks such as bottleneck locating and capacity measuring, making this benchmark tool quite practical. That said, we are still actively working on adding more operations, but only in a way that will keep the tool simple, practical, and universal.

4.2 Usage Patterns

To simulate diverse usage patterns, our workload model can be flexibly configured in terms of *concurrency patterns*, *access patterns*, and *usage constraints*. In concurrency patterns, one specifies the number of clients that should be simulated by the workload generator. This directly enables people to evaluate the scalability of a system against rising workloads. In access pattern, one defines a set of operations that each client should issue to the target system. According to our design, each operation can be customized in terms of container path, object path, object size[1], and operation ratio. As can been seen, the access pattern lends people a lot of possibilities for designing their workloads: it can be read-intensive, write-intensive, or hybrid; it can deal with small objects, large objects, or both; and these objects can also be spread among many or only a few containers. It is conceivable that such richness of the workload design space can benefit not only people who want to evaluate and compare different object storage systems, but also those who want to probe specific systems for an in-depth analysis. The last aspect of the usage patterns involves various usage constraints. They can be expressed as total running time of a workload, total number of operations to be submitted, or total number of bytes to be transferred. If more than one such limit is presented, each of them can individually lead to the exit of a running workload.

4.3 Workload Model

In order to make the workload model even more flexi-

[1]Note that not every operation requires the size of an object to be specified, such as read or remove operations.

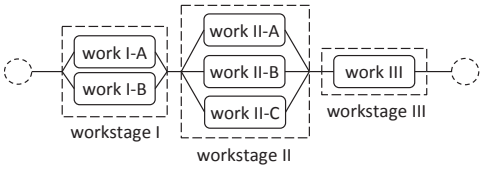


Figure 5: Workload modeled as a workflow

ble, we have employed workflow. In our model, a workload is modeled as a *workflow* consisting of multiple *workstages*, with each workstage representing a step in the workflow and focusing on a particular activity. For example, a workflow can have three different workstages: an initial stage, a main stage, and a cleanup stage. The initial stage can be configured to setup the benchmark environment (e.g., by uploading certain objects and creating dedicated containers). The main stage is responsible for launching the real workload. Finally, the cleanup stage removes all the data involved in this workload, restoring the target system to its original state. When a workflow is executed, each workstage will be executed in sequence. A workload is marked finished only after all its workstages have been completed. Failure of any workstage in the middle will terminate the overall workload execution immediately.

In order for a workstage in our model to perform multiple independent tasks simultaneously, a workstage is further regarded as a group of *works*, with each work representing a special job within a workstage. For example, a workstage can be defined to comprise two different works: one mimics a web application, frequently issuing read-intensive operations on small objects, while the other pretends to be an archive program constantly streaming up large objects. When a workstage is executed, all of its internal works are executed at the same time. A workstage is considered completed only after all its works have been completed. If an error occurs in any one of these works, the remaining works will be automatically aborted and the overall workstage will be deemed to have failed.

Works are atomic and are decoupled from each other. Each work can enjoy its own configuration context including its usage patterns, user credentials, and even the target system. As a result, it is possible for one to simulate multiple users at the same time, as well as to statically perform load balancing among different system portals. At runtime, each work is executed by a certain number of *workers*. A worker is a logic concept of a client; the number of workers is derived from the concurrency patterns associated with the work. The overall workflow model is illustrated in Figure 5.

5. BENCHMARK IMPLEMENTATION

We have developed a tool named COSBench to execute the workflows described in the previous section. In order to make sure that our tool can work with various object storage systems, we have cautiously designed this software so that people can easily adapt our storage interface to their specific implementations. Attention has also been paid to ensuring the scalability of the tool to sustain large numbers of clients. In this section, we describe the system architecture of our tool and highlight its extensibility and scalability.

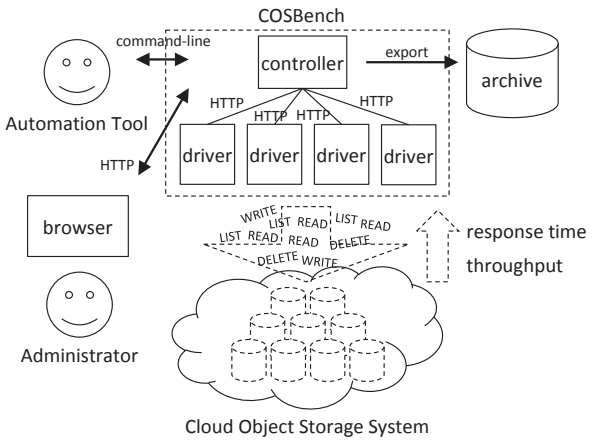


Figure 6: COSBench system overview

5.1 System Architecture

The COSBench is a web based Java application capable of generating workloads to a given object storage system and measuring the performance this specific system exhibits. As is illustrated in figure 6, our tool consists of two kinds of components: the *controller* and the *driver*. The controller serves as the facade of COSBench: it provides interfaces through which users can operate and control the benchmark tool as a whole. When users submit their workloads to the controller, internally, the controller divides these workloads into multiple tasks and schedules these tasks to different driver instances as *missions* for execution. As such, the controller communicates with drivers through the web interfaces they expose to assign missions to them or to retrieve execution status and results from these drivers. At the driver side, in order to execute a mission, an *auth* agent is first created to perform the required user login. After that, a certain number of *worker* agents are spawned to stress the target storage system in three phases: the *ramp-up* phase, the *transaction* phase, and the *ramp-down* phase. Drivers are independent from each other as they receive commands only from the controller, not from any other driver instances. The controller is responsible for synchronizing the execution of different missions associated with different drivers, ensuring the integrity of a workload. During the execution of a workload, the controller constantly polls related drivers to gather runtime status of its missions. When a workload is finished, the controller aggregates the performance data retrieved from those outsourced missions and archives the final report into a persistent place.

In order to facilitate comprehensive performance examinations, our final report comprises average, 90th, 95th, 99th, and 100th response times, throughput, bandwidth, success ratio, and a detailed latency histogram for each operation in each stage of a workload. In addition, a time series plot of the performance readings from ramp-up phase to ramp-down phase is also provided for each operation in each stage. To secure ease-of-use, we have made our tool accessible from both web browsers and command-line utilities, making it friendly not only to human users, but automation tools as well. Users can use these interfaces to submit or cancel workloads, check runtime status, exam or export final reports.

5.2 Extensibility

Several efforts have been made to ensure the extensibility of our tool. Firstly, we have made user authentication a dedicated interface, independent from the storage. According to our experience, it is possible for an object storage system to support multiple user authentication protocols (e.g., both Swauth [14] and Keystone [7] can serve as the authentication mechanism for Swift). One the other hand, it is also possible for multiple object storage systems to share the same authentication protocol (such as OAuth [9] and SAML [13]). By recognizing separate interfaces for both authentication and storage, our tool not only supports convenient switch among different authentication alternatives, but encourages software reuse as well.

Another effort lies in the workload configuration. In our tool, a workload is configured in an XML file prepared by a user when submitting a workflow. Unfortunately, difficulties arise when we try to design the schema of such XML. Since different object storage systems have varying definitions of their operation parameters, it is therefore impractical and infeasible for us to accommodate all possibilities in our workload configuration. As an example, people can specify certain "policies" when creating containers. In some systems, people must set a strategy that controls erasure code. For others, distinct options should be supplied to control various features provided by the system, such as container-to-container synchronization, object versioning, and others. To assure that our configuration can adapt to these manifold parameters, we regard all parameters as key-value pairs and merge all of these pairs into a single field which we call `config`. For example, a write operation might be configured as follows.

<operation type="writ", ratio="30",
config="containers=u(1,100); objects=u(1,100);
sizes=u(64,128)KB; object-expiration=10" />

As can be seen, there are four configuration pairs defined: `containers`, `objects`, `sizes`, and `object-expiration`. While the former three (which define the ranges and sizes of the containers and objects involved) are core to all implementations and are thus required by the COSBench, the last one is of a foreign nature and is forwarded by the COSBench to the adaptor for further interpretation and handling. With different adaptors responsible for defining and checking their own parameters within this unified framework, we successfully decoupled our configuration schema from various specific protocols, therefore achieving not only extensibility but flexibility as well.

Finally, the modular design of our system also considerably facilitates the overall extensibility of the tool. To be more specific, the COSBench is built upon Eclipse Equinox [4], an OSGi [11] implementation that supports dynamic module management. Layered above this infrastructure, our system consists of multiple modules known to the infrastructure as bundles. Bundles can expose services to a central service registry or import services from it. At runtime, when bundles are launched, the service registry will automatically map service providers (bundles that expose services) to their consumers (bundles that import services), allowing service consumers to use services without the knowledge of the providers. To add new storage types into the COSBench, people create new bundles, implement the storage interface inside their bundles, and configure their bundles to expose

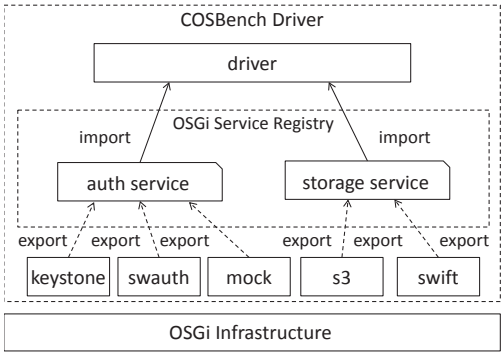


Figure 7: Modular design of the COSBench

their implementation as services. At runtime, once system is booted, all exposed storage services will be automatically bound to the driver. (The driver bundle is configured to import all available storage services). When a mission (Section 5.1) is assigned, the driver will iterate over all storage services bound to it to see if any of them is capable of handling the incoming mission. If such service is found, this service will be invoked, creating a new adaptor instance, which will then be attached to this particular mission for execution. Otherwise, the mission will be rejected. The relationship among the driver, adaptors, and service registry is depicted in Figure 7. As adaptors are decoupled from the driver and are dynamically plugged into the system at runtime, new adaptors can be independently developed, configured, and integrated without modifications (or awareness) of the core system.

5.3 Scalability

One important issue regarding characterizing web-scale storage systems is that the benchmark tool should be highly scalability so that a large number of clients can be simulated to place a substantial pressure on the target system. In our tool, as workloads are executed at the driver side, the overall throughput of the tool is therefore proportional to the number of driver instances being provisioned: more drivers instances means more CPU cores and more bandwidth, and thus more clients the tool is able to sustain. Besides, within each driver, workers are also carefully kept isolated from one another. Each worker has a dedicated execution context including a connection to the target system, a random number generator, various counters for performance measurements, as well as other status variables required. As such, workers can be executed in a fully parallel manner, which leads to the high scalability of COSBench.

6. EXPERIMENTS

The latest stable release of the COSBench is 2.0.0-GA. Internally, we have been using this tool to support our characterization on multiple object storage setups we deployed in our lab. While some of these systems are built with open source technologies, others are obtained directly from the commercial worlds. So far, we have found it quite smooth for us to write adaptors for different object storage systems. Besides our internal uses, the COSBench is also shared among different parties with various backgrounds both inside and outside Intel: some are object storage engineers

Table 2: Workloads used in our experiments

Workload	Read/Write Pattern	Object Size
Workload A	Read	128 KB
Workload B	Write	128 KB
Workload C	Read	10 MB
Workload D	Write	10 MB

who employ workloads to better verify their system designs; some are decision makers who are responsible for gauging and weighing different object storage offerings; and some are cloud builders who need data to promote their solutions. Of course, there are also pure researchers just like us.

In this section, we report the results of the experiments we have performed on two Swift setups under four distinct workloads. As our intention is to obtain a deep understanding of different object storage systems and insights to guide optimization as well, more have to be conducted before we can be rewarded with such knowledge. For this, we are still actively performing various experiments in order to reach our targets. We plan to share our further findings in a separate report where more data will be collected and more systems compared.

6.1 Workloads

We pay attention to the performance, scalability, and resource usage patterns exhibited by different systems under distinct read-write patterns and object sizes. The workloads are illustrated in Table 2. Although in the real world no application ever displays such simple and naive patterns, these workloads in fact suffice to serve as the starting points towards understanding a storage system. Besides, the baseline performance revealed under these workloads can help channel our future research efforts to the most compelling parts. As our ultimate goal is to disclose various factors throttling the performance of a system and to propose ideals for resolving such bottlenecks, simple and pure workloads can actually better help us to achieve this specific goal.

In our experiments, for read-only workloads, the target system were prepared with 1 user account, 100 containers, and 10,000 objects. For write-only ones, the systems were prepared similarly except that no objects are pre-inserted. To examine scalability, we varied the worker number from 5, 10, 20 ... to 2,560 (increased exponentially). We repeated each test case with three separate runs (each consisted of a 300s ramp-up phase and a 300s transaction phase) and took the average as the final result[2].

Besides the results reported by the COSBench, we also collected various hardware status monitored by utilities such as `iostat`, `vmstat`, and `sar`. This was performed by an automation tool we had separated developed with which the COSBench was integrated. The integration in fact took trivial efforts, thanks to the command-line interface exposed by the COSBench (Section 5.1).

6.2 Experimental Setup

For our experiments, we built two independent Swift setups on 22 server-class machines with each cluster enjoying five client nodes, five storage nodes and one proxy node.

[2]In the beginning, it was five runs and 500s for both phases. However, we later found that three runs and 300s were fine for both setups (saving us 64% of the total execution time).

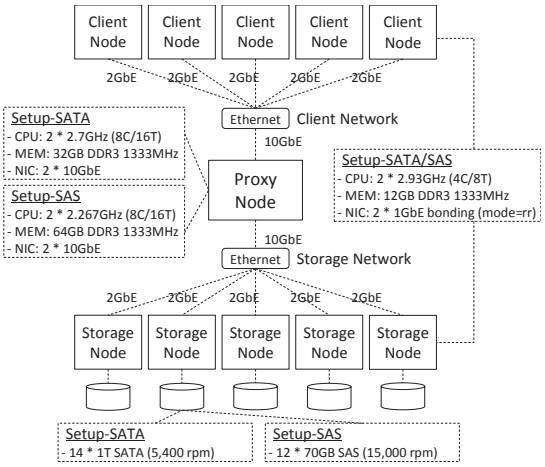


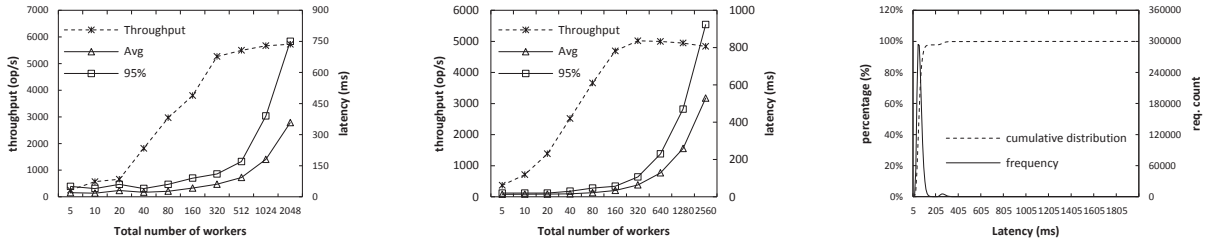
Figure 8: Hardware configurations

The client nodes ran the COSBench, with each running a `driver` instance and a special one running an additional `controller` instance. The storage nodes and the proxy node were all used to run Swift, with the proxy node serving as a front-end controller forwarding requests to back-end storage nodes where specific operations were carried out. As such, we ran a certain number of `proxy-server` instances on the proxy node, and a certain number of `object-server`, `container-server`, and `account-server` instances on each storage node. Although in production environments other services, such as `object-replicator`, `object-updater`, `object-auditor`, to list just a few, should also be launched on storage nodes to keep data consistent and durable, we turned these services off to render less variance to system's performance[3]. When our experiments were performed, the latest release of Swift was 1.4.8. As Swift is still under heady development, increases or changes are both likely to be seen in the performance of its future releases. However, as we share our data mainly to demonstrate the value of our tool in facilitating performance comparison and system optimization, our data are not that meaningless.

As can been seen from Figure 8, there were a few differences between the above two setups: Setup-SATA possessed a more powerful proxy node whereas Setup-SAS was attached with more sophisticated disks. We will review the impacts of such hardware discrepancies when we discuss the results of our experiments. Note that in both setups, the aggregate bandwidth from client nodes to the proxy node was made equal to that from the proxy node to storage nodes, suggesting that our clients were able to consume as much throughput as the storage system could produce. We also allocated separate disks to run operating systems and a separate network for system control, protecting our environment from unintended noises.

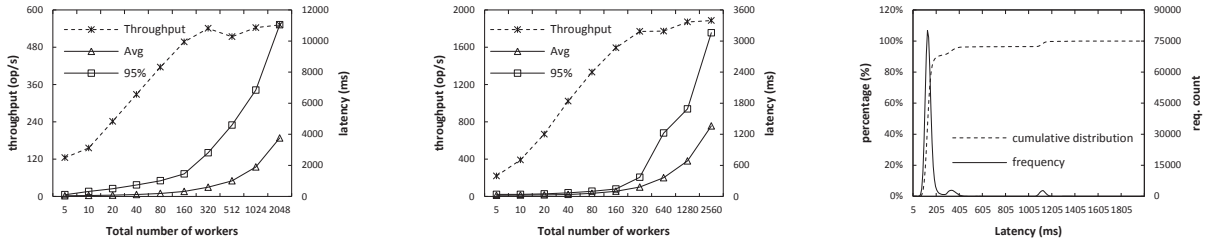
For both setups, we tried our best to tune our systems to reach maximum performance. Since our setups were built upon different hardware infrastructures, each setup required different configurations to achieve its optimal performance. We share some of our efforts in Section 7.

[3]According to our experience, these "background" services will produce haphazard impacts on the overall system, resulting in as much as 18% of variance of performance among multiple runs.



(a) Setup-SATA (b) Setup-SAS (c) latency histogram of Setup-SAS under its peak performance

Figure 9: Results of workload A. In Setup-SATA, the performance peaked at 320 workers (5,266 op/s in throughput, 110 ms in 95% latency); in Setup-SAS, 320 workers (5,019 op/s in throughput, 106 ms in 95% latency).



(a) Setup-SATA (b) Setup-SAS (c) latency histogram of Setup-SAS under its peak performance

Figure 10: Results of workload B. In Setup-SATA, the performance peaked at 320 workers (540 op/s in throughput, 2,810 ms in 95% latency); in Setup-SAS, 320 workers (1,769 op/s in throughput, 370 ms in 95% latency).

6.3 Workload A: Small Object Reads

The results of workload A are illustrated in Figure 9, which shows the throughput, average, and 95% response time observed under mounting number of workers. For both setups, with more workers stressing the system, response time increased as throughput increased. Besides, both setups exhibited good predictability in their response times. As can be seen from Figure 9-c, most requests were completed somewhere between 35 ms and 75 ms in Setup-SAS, suggesting that the variance among different requests was quite low.

Although Setup-SATA ran on slower disks, it actually exhibited 5% more throughput than Setup-SAS, which led us to conjecture that it was the CPU at the proxy node that mattered most to the performance of these systems under this workload. This was later ascertained by the exorbitant CPU utilization found in both setups in particular under their peak performance. After all, sustaining over 5,000 requests per seconds does require a nontrivial amount of computation power.

To further investigate scalability, we varied the number of storage nodes from one to five in Setup-SAS and obtained the throughput versus worker curve for each different setting. The results are showed in Figure 11. Note that when one single storage node was provisioned, we configured Swift to maintain only one copy for each object uploaded. If there were two storage nodes, the number of copies maintained was increased to two. Otherwise, the number was set to three, the default one. Note also that in normal cases Swift only takes one copy to serve a read request (more copies will be consulted only if all previous attempts have failed). As can be seen from the results, the throughput increased as the number of storage nodes increased until the proxy node became the bottleneck and prevented the system from gaining any further capacity. This could have be alleviated by provisioning more proxy nodes to lend more CPU power to the proxy tier. In our specific case, it is quite clear that every three storage nodes should be matched with one proxy node, which indicates that we should have provisioned two proxy nodes rather than a single one.

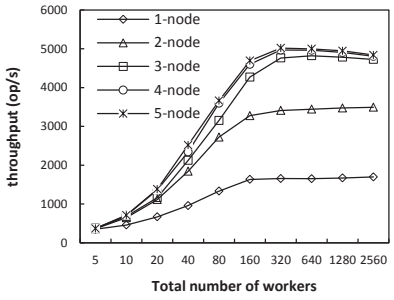


Figure 11: Results of the scalability test in Setup-SAS under workload A.

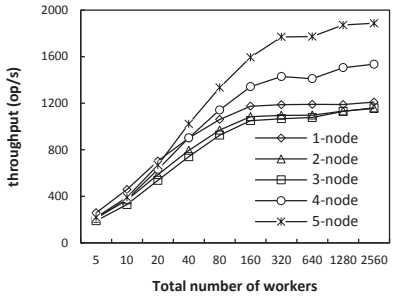


Figure 12: Results of the scalability test in Setup-SAS under workload B.

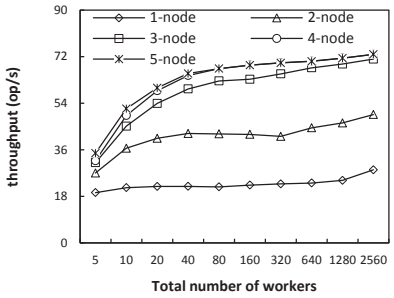


Figure 13: Results of the scalability test in Setup-SAS under workload C.

6.4 Workload B: Small Object Writes

Workload B shows a different picture from workload A by focusing on the write performance instead. The results are provided in Figure 10. This time, Setup-SATA presented a much worse performance than Setup-SAS in that the 95% response time curve in the former was noticeably steeper and that the throughput of it was also terribly lower. As Setup-SATA was not so fortunate to be blessed with such nice disks as the ones given to Setup-SAS, it followed that the poor performance exhibited by Setup-SATA was due to its limited disk power. This is within one's expectation since each object uploaded to Swift will cause three copies of data to be created and three replicas of container metadata to be updated as well. As a consequence, there are more works left for disks than for the CPU. To convince ourselves, we replaced all storage disks used in Setup-SATA with SSD disks. Results showed that the throughput increased significantly from, for example, the original 550 op/s to 1,800 op/s under 2,048 workers, which confirmed our above presumption that the root cause of the poor performance in Setup-SATA was its slow disks.

As can be seen from Figure 10-c, Setup-SAS achieved a slightly lower predictability in its response times than it did in workload A, with most requests completed in 85 ms to 175 ms under its peak performance this time. After all, each write operation has to undergo six internal round trips with three for storing object copies and the other three for updating container, which inescapably introduces more variance to the overall latency.

A similar scalability test for Setup-SAS was performed, with the results plotted in Figure 12. The throughput initially went down slightly as the number of storage nodes rose from one to three. This however was understandable since the number of copies that must be created for each object uploaded increased as well. As the number of storage nodes kept rising from three to five, a proportionate increase of the throughput was observed, indicating that the workload was well balanced among the backend storage nodes.

6.5 Workload C: Large Object Reads

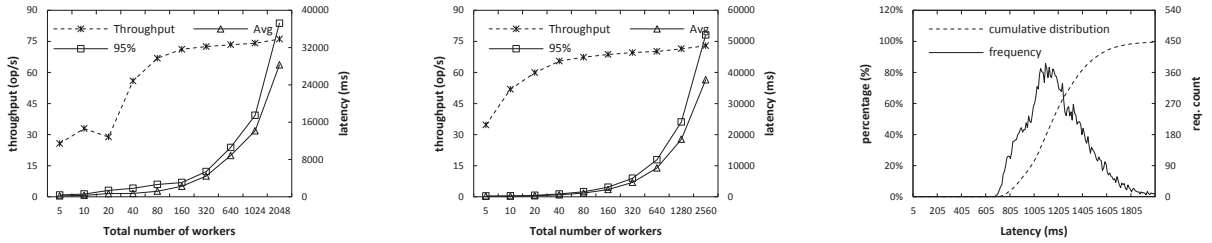
Workload C intends to evaluate the ability of a system to send out large objects. The results are illustrated in Figure 14. Sadly, neither setup exhibited as good predictability in response times as they had achieved in previous workloads. Figure 14-c shows that individual response times observed in Setup-SAS under its peak performance spanned a long range from 650 ms to 1,900 ms, indicating that the bandwidth witnessed by different workers varied a lof from one request to another.

Since the performance of our two setups was similar to each other in this workload, we assumed that the bottleneck should be neither the disk nor the CPU. After a careful scrutiny, we realized that the key to further optimization actually resided in the client network. Although our aggregate bandwidth between storage nodes and client nodes should be as large as 1,250 MB/s (10Gb/s) in theory, we only got 700 MB/s at most in practice. We later found that this was due to the overhead incurred by bonding network links together. In our case, after two 1Gb links were bonded on each storage node and client node, the new link can only receive data at a low speed of 140 MB/s (56% of the theoretical maximum). As a consequence, our five client nodes could only read data at a maximum speed of 700 MB/s, thus rendering throughput an upper bound of 70 op/s. To resolve this issue, two different approaches were taken. In the first one, we provisioned five more client nodes to the setup with each node adding 140 MB/s more bandwidth to the aggregate bandwidth. In the second one, we set up four virtual machines on each client node using SR-IOV instead of bonding (with each VM assigned a 1Gb SR-IOV NIC). The results showed that both approaches were able to improve network bandwidth to its theoretical capacity; the throughput increased from 700 MB/s to over 1,100 MB/s.

Figure 13 shows the results of the scalability test under this workload in Setup-SAS. The throughput increased as the number of storage nodes increased, until the network became the bottleneck. If the network issue had been resolved, we would have seen further increases in throughput proportionate to the number of storage nodes beyond three. Another thing worth mentioning here is that our proxy node was capable of sustaining more storage nodes (at least five nodes) in this workload than it was in workload A (three nodes at most), which constitutes a concrete example of how workload characteristics affect system configuration and performance tuning.

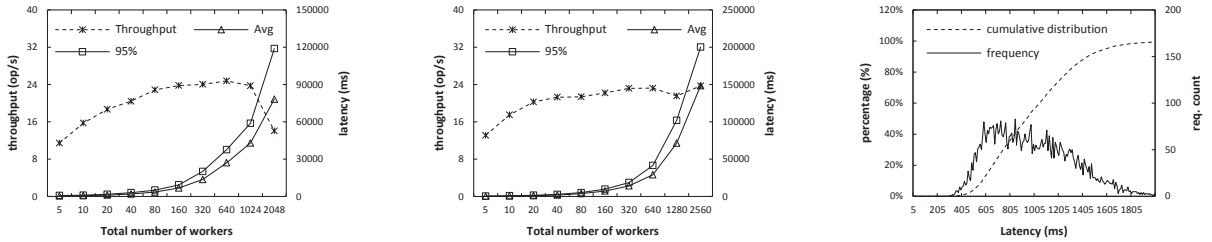
6.6 Workload D: Large Object Writes

Finally, the results of workload D are illustrated in Figure 15. Unfortunately, both setups exhibited very poor predictability in their response times. As can been seen from Figure 15-c, which shows the latency histogram of Setup-SAS under its peak performance, the response times almost



(a) Setup-SATA (b) Setup-SAS (c) latency histogram of Setup-SAS under its peak performance

Figure 14: Results of workload C. In Setup-SATA, the performance peaked at 160 workers (71 op/s in throughput, 3,060 ms in 95% latency); in Setup-SAS, 80 workers (67 op/s in throughput, 1,636 ms in 95% latency).



(a) Setup-SATA (b) Setup-SAS (c) latency histogram of Setup-SAS under its peak performance

Figure 15: Results of workload D. In Setup-SATA, the performance peaked at 80 workers (23 op/s in throughput, 5,100 ms in 95% latency); in Setup-SAS, 20 workers (20 op/s in throughput, 1,569 ms in 95% latency).

evenly distributed all over the time line from 500 ms to 1,600 ms.

Since the throughput exhibited by both setups in this workload happened to be a third of the throughput they exhibited in the previous one, this led us to believe that the key performance issue for this workload was still at the network layer, and that the high contention of the network bandwidth should take the blame for the high variance in the response times we observed. Note that in this workload, the data actually flew from the clients to the storage nodes, so it was the storage nodes that were reading the data and thus suffering from the poor receiving performance caused by network bonding. As such, it was legitimate to say that the throughput would be improved if more storage nodes had been provisioned, and worsen if less. In fact, the scalability test we performed later yielded evidences which could support our above assumption. As can be seen from Figure 16, the throughput increased linearly to the number of storage nodes as the latter rose from three to five. We actually also checked the status of both the CPU and the disks on storage nodes; however, no symptom of bottleneck had been found in these hardware resources. As a result, it is safe to take the storage network as the culprit for inhibiting scalability.

7. EXPERIENCES

As we mentioned early, we tried our best to tune both our setups to achieve their maximum performance. Since our two setups were associated with different hardware configurations, each of them had to be considered and optimized individually. For example, compared with Setup-SAS, we found that more server instances (`proxy-server`, `object-server`, and the like) have to be launched in Setup-SATA in order to match and compensate its slow disks. However, attempting to launch an equal number of instances in Setup-SAS as in Setup-SATA would only do a disservice to the performance of the former. As a lot of efforts have been taken to better configure our systems, in this section, we share some of our experiences by discussing two factors that, we think, people should take into consideration when tuning their systems.

Firstly, one should make sure that the hardware has been appropriately configured. Ideally, each element of the system should be kept balance with the rest to preclude any explicit bottleneck. For example, in our early experiments, the proxy node in Setup-SATA only had one single 10Gb NIC port configured, which was connected both to the client network and the storage network. We later realized that the system was ill balanced since this specific port turned out to be a bottleneck. Once a separate port was enabled, the performance immediately improved from 3,893 op/s to 4,726 op/s in the case of workload A. Besides watching the hardware, it is also important to be familiar with the internal logic of a system. We find that this kind of knowledge can help people better understand the behavior of a system and better read test results as well. For example, in our experiments, we initially configured all our workloads to use just one single container. The performance we observed in work-

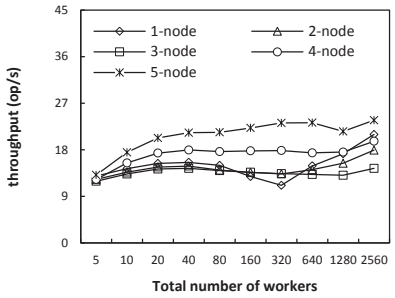


Figure 16: Results of the scalability test in Setup-SAS under workload D.

load B under this setting was only 197 op/s for Setup-SATA. Such low performance happened to remind us of one important detail we had learned from the source code: when each object is uploaded, besides creating three copies, Swift will additionally update the corresponding container to register this newly inserted object. Since the status of each container is persisted in a dedicated sqlite DB file with three replicas, the poor performance was very likely to be caused by the lock contention often associated with concurrent database modifications. To resolve this issue, we tuned our systems as follows: (i) we configured our workloads to use 100 containers instead of a single one to alleviate lock contention; then (ii) we increased the number of `container-server` instances launched in both setups to improve responsiveness; and (iii) we allocated dedicated SSD disks to hold all container DB files so as to work around the limitation of the HDD disks (this was for Setup-SATA only). Results showed that each of these actions was able to individually enhance performance. They collectively achieved a 280% increase in throughput in the case of workload B for Setup-SATA, from the previous 197 op/s to what we listed in Section 6.4.

8. RELATED WORK

Object storage systems are no longer strangers to developers, with the proliferation of these systems observed during recent years. Some object storage systems (such as Walrus in Eucalyptus [5] and Cumulus in Nimbus [8]) are implemented as immediate adaptor over traditional storage technologies. Some (such as Swift in Openstack [10]) are designed from bottom up with efforts devoted to ensure scalability, durability, and cost-effectiveness. Besides, there are also ones (such as RADOS [30] and HYDRAstor [23]) that are intended mainly to support file systems, and ones (such as Windows Azure Storage [20] and Yahoo! Walnut [21]) that are built with the goal of creating a unified enterprise storage infrastructure which can sustain better flexibility, manageability and application performance. Last but not least, one can also adopt object storage by using cloud services offered by Amazon [2], Rackspace [12] or Google [6].

The ideal of creating a benchmark tool to evaluate the performance of a system is not unusual. Efforts of implementing such a tool have already been seen in the newly-emerged area of object storage. For example, there is a benchmark utility included in the distribution of Swift, which is called SwiftBench [15]. Our tool is similar to it as both can be used to characterize Swift setups. However, while the former is dedicated for Swift to facilitate its system development, our tool is designed to be a general benchmark tool and is intended for a variety of purposes including system comparison, tuning, and optimization. Another example of such efforts is Haystress [18], which is provided to support the performance testing of Haystack, which is in turn a distributed photo repository used exclusively inside Facebook. Our tool is similar to Haystress since both can simulate different kinds of object storage applications. However, our tool is built with a more sophisticated workload model and is therefore capable of generating more diverse usage patterns. Besides these serious programs, scripts written by perl or curl have also been witnessed in the characterization of object storage systems. Although scripts are easier to implement and are not bound to pre-designed workload models, they lack the extensibility to work with different systems and various features to ensure ease-of-use. To this regard, we have designed our tool with attention paid on both issues, thus allowing our users to focus more on their innovation rather than benchmark details. Besides, our work also resembles YCSB [22] though we focus on object storage systems while they "nosql" data stores.

One prototype version of our tool was presented in a previous short report [32]. Compared with this release, our current version has been fully revised with a stronger system architecture which gives birth to the high extensibility and scalability of the tool. In addition, the workload model as well as the user-interface accompanying the tool has also been considerably improved in our current distribution, making the tool more practical and easier to use.

9. CONCLUSION

We have presented a benchmark tool we named *Cloud Object Storage Benchmark*. This tool is developed with the main intention of helping people better evaluate and compare different object storage systems. It is also designed to facilitate system tuning and optimization. To demonstrate the effectiveness of our tool, we have used it to characterize the performance of two Swift setups under four distinct workloads, and discussed the impacts of different hardware configurations on system performance. We also showed how hardware profiling data can be utilized to identify performance bottlenecks and guide system optimization. As our tool can be flexibly adapted to other object storage systems, people can easily use this tool to characterize their own systems, either for facilitating performance comparison or for driving system optimization.

COSBench is still under our active development and is being enhanced in many aspects. One important part of our future work is to make the tool open source, so that it will be easier for people to use this tool and to extend it with their own storage adaptors. Besides, we are also working on adding additional features into this software. For example, one feature that we are implementing can empower people to detect possible data corruptions: it can be enabled by configuring the tool to generate and check the MD5 value of each object it handles. Another track of improvements concerns ease-of-use. For example, we are implementing a web interface which enables our users to create workload configurations directly in their browsers. As a result, this "online web editor" can free our users from the burden of expressing their usage patterns in raw XML.

10. REFERENCES

[1] 905 billion objects and 650,000 requests/second. http://aws.typepad.com/aws.

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

[3] Cdmi. http://www.snia.org/cdmi.

[4] Equinox. http://www.eclipse.org/equinox.

[5] Eucalyptus. http://www.eucalyptus.com.

[6] Google cloud storage. http://www.google.com/enterprise/cloud.

[7] Keystone. http://docs.openstack.org/developer/keystone.

[8] Nimbus. http://www.nimbusproject.org.

[9] Oauth. http://oauth.net.

[10] Openstack. www.openstack.org.

[11] Osgi. http://www.osgi.org/Main/HomePage.

[12] Rackspace cloud files. http://www.rackspace.com/cloud.

[13] Saml. https://www.oasis-open.org/committees/security.

[14] Swauth. https://github.com/gholt/swauth.

[15] Swift bench. https://github.com/openstack/swift.

[16] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *FAST*, 2009.

[17] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *FAST*, 2004.

[18] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. In *OSDI*, 2010.

[19] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD*, 2008.

[20] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[21] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. In *SIGMOD*, 2012.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[23] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: a scalable secondary storage. In *FAST*, 2009.

[24] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *USENIX ATC*, 2012.

[25] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC*, 2008.

[26] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *USENIX ATC*, 2000.

[27] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system. In *FAST*, 2010.

[28] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, 2012.

[29] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI*, 2006.

[30] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW*, 2007.

[31] N. Yezhkova, R. L. Villars, L. Conner, and B. Woo. Worldwide enterprise storage systems 20102014 forecast: Recovery, efficiency, and digitization shaping customer requirements for storage systems. *IDC*, 2010.

[32] Q. Zheng, H. Chen, Y. Wang, J. Duan, and Z. Huang. Cosbench: A benchmark tool for cloud object storage services. In *CLOUD*, 2012.