Towards Optimized Self-Management of Distributed Object Storage Systems

Roman Talyansky[*], Ewnetu Bayuh Lakew[†], Cristian Klein[‡], Francisco Hernandez-Rodriguez[‡], Erik Elmroth[‡], and Eliezer Levy[†]

Abstract. Cloud storage is increasingly adopted by users due to simplified storage systems compared to on-premise storage. These systems are mostly presented as Object Storage Systems (OSSs), hiding issues, such as redundancy, from users. As new industries are considering adopting clouds for storage, OSSs have to evolve to support new needs. Among the most challenging is assuring guaranteed performance.

In this paper, we present Controllable Trade-offs (CTO), an OSS-agnostic solution to add performance guarantees. CTO presents itself as a thin layer that mediates requests between the user and the OSS. For generic support, performance is controlled by tuning the rejection probability, and implemented as a user-side queue. Results show that CTO may reduce penalties 3.23 times on average and up to 68 times when the load is high.

1 Introduction

Individual users and businesses are increasingly adopting clouds for simplifying their data storage needs. Cloud storage platforms are mostly designed as **Object Storage System (OSS)**, allowing users to get and put objects (i.e., whole files), often through a simple REST interface [11]. They thus save the user from dealing with low-level details such as location, redundancy, backups and load-balancing. Cloud storage has attracted considerable attention: One may find both commercial (Amazon S3 and Glacier [30], Google Storage [2]) and open-source implementations (OpenStack Storage [19]).

As new industries are considering moving their data into the cloud, OSSs need to evolve to support new needs [15]. One of the greatest concern is the fact that

[*]Huawei, Israel, email: {eliezer.levy,roman.talyansky}@huawei.com

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {ewnetu, francisco, elmroth}@cs.umu.se

[‡]SimScale GmbH, Germany, email: cklein@simscale.com.

existing OSSs are void of any performance guarantees (c.f., [1]). In fact, with existing cloud storage solutions, users may observe unpredictable Key Performance Indicators (KPIs), such as response times and throughput, that are highly sensitive to the load of the platform. Hence, a solution is needed that enforces KPIs. This would allow providers to complement their Service Level Agreement (SLA), i.e., the contract signed between the user and the provider, with performance guarantees.

Unfortunately, it is unlikely that these SLAs can always be kept. For cost efficiency reasons, cloud storage providers must operate close to their peak performance. This means that in case of failure or unexpectedly high load, one or more SLAs have to be violated. Hence, a further requirement would be for the OSS to differentiate among *classes* of users (e.g., gold, silver, bronze), as stipulated in the SLA, and to minimize the penalties that are paid, should the system be overloaded.

Enforcing SLAs for storage has long been seen as a great challenge, mostly due to the black-box nature of the underlying storage platform. For example, issues such as the elevator algorithm, hybrid (mechanical and solid-state) drives and RAID make performance highly unpredictable. Therefore, most approaches resort to providing only differentiated SLAs, in which higher classes get more performance than lower ones, without dealing with guaranteed SLAs, e.g., a gold user gets exactly 100 MB/s throughput.

In this paper we propose Controllable Tradeoffs (CTO) a front-end to OSSs that mediates requests to enforce differentiated and guaranteed SLAs. CTO works by monitoring the client requests, analysing the number of concurrent readers, writers and their average think time, and tuning the probability of rejecting a request. Expected KPI values are estimated based on off-line calibration. The highlight of our approach is that the solution is back-end agnostic.

Our contributions is three-fold:

1. We translate the goal of ensuring differentiated, guaranteed SLAs to a penalty-function minimization problem (Section 3).

2. We describe CTO, a back-end agnostic solution (Section 4).

3. We evaluate the approach on three back-ends: a machine with a single Hard Disk Drive (HDD), a server with HDDs in RAID10 and a distributed storage cluster (Section 5).

Results show that CTO may reduce penalties 3.23 times on average and up to 68 times during high load.

2 Related Work

Numerous research efforts have been devoted to improve performance and provide differentiated services for packet switched networks [14, 18, 24], web servers [7, 21] and compute clouds [20, 28, 35]. However, all these works are not suitable for storage systems due to their unique characteristics [10]: Their peak performance is highly sensitive to the workload, e.g., sequential vs. random access, read vs. write.

The problem of differentiated service for storage systems can be addressed from two angles: fine and coarse granularity. Most of the works focus on providing performance guarantees at a fine granularity for a single storage system, using performance models of a disk system to estimate seek and rotational delays for each I/O request. Examples of such works include guaranteeing performance for disk I/O [33], configuring storage systems to meet performance goals [34], allocating storage bandwidth to application classes [27], mapping administrator specified goals to appropriate storage device actions [29], modifying OS block layer to support classes [23] and including classes in I/O drivers [9]. However, due to their focus on low-level storage system scheduling, it is impractical to scale them to large-scale deployments as required in cloud storage. Moreover, using these approaches, it is very difficult to translate high-level objectives, such as reducing overall penalty, into low-level semantics.

Storage service differentiation based on proportionate I/O bandwidth allocation have also been developed for storage systems [16, 17, 26, 31] that operate at a coarse granularity, e.g., average response time per class. However, the focus of these works is on providing proportional I/O bandwidth allocations based on pre-configured weights for each class. As a result, observed performance of different classes may degrade during system overload. In contrast, our goal is to offer performance guarantees for higher classes and only degrade lower classes in case of overload, as required to minimizing the penalty.

3 Problem Definition

In this section we give a more precise definition of our problem. We consider an OSS, that is concurrently accessed by multiple users, each reading (get) or writing (put) objects. Each user has an SLA associated with her account, that stipulates target values for KPIs, revenues made by the provider in case of successful delivery of the service and penalties paid to the user in case the target performance could not be kept. On the provider's side, a *Utility Function (UF)* is configured to map user performance to expected profits, taking into account factors such as revenues, penalties, user retention and Total Cost Ownership (TCO) of hardware and software.

We are looking for a *back-end agnostic* solution to optimize provider profits over an existing OSS. Indeed, given the plethora of existing OSSs and the complexity they handle, it does not make sense to write a new one. Thus, we are aiming for a generic front-end component, that *mediates* messages between the user and the OSS to maximize provider profits as conveyed by the UF.

Ideally, the system should behave as follows. If the OSS is not overloaded, then each user should get at least the performance agreed in the SLA. In other words, the system may be work-conserving, redistributing spare throughput to improve user performance, thus encouraging new users to join the service. If the OSS is overloaded, then the system should try to minimize paid penalties as required to maximize profit.

4 Solution

This section describes Controllable Tradeoffs (CTO) our solution for guaranteed performance over Object Storage System (OSS). First we give an overview of CTO's architecture, then we present each component. In particular, we detail the estimator function, the core algorithm of CTO.

4.1 Architecture

The architecture of CTO is depicted in Figure 1. It features a thin layer, that mediates requests between the user and the OSS, and an autonomic feedback loop [4], composed of the classical monitor, analyze, plan and execute phases. Monitor and analyze is done by the *monitoring* component, that is responsible both for intercepting the user requests to the OSS and analysing the workload. It extracts workload information (noted W) such as the number of users, the average time between consecutive requests, called Think-Time (TT), and the average request size. Planning is done by the *utility function*, *estimator* and *optimizer* that use information about the workload to devise a plan. Finally, execution is done by the *control knob* which modulates the requests to enforce the plan.

To understand our contribution, we first need to detail the choice of control knob. Then we can highlight how the utility function, estimator and optimizer achieve system goals.

4.2 Control Knob

The purpose of the control knob is to allow an OSS-agnostic way to control observed performance of individual users. This is achieved by artificially delaying the processing of requests. Depending on how the delaying is achieved, one may differentiate between system-delay knobs and user-delay knobs. *System-delay* knobs enforce user request delays within the storage system, e.g., in system queues. Delay is achieved by forcing users to wait for an additional amount of time until their requests complete.

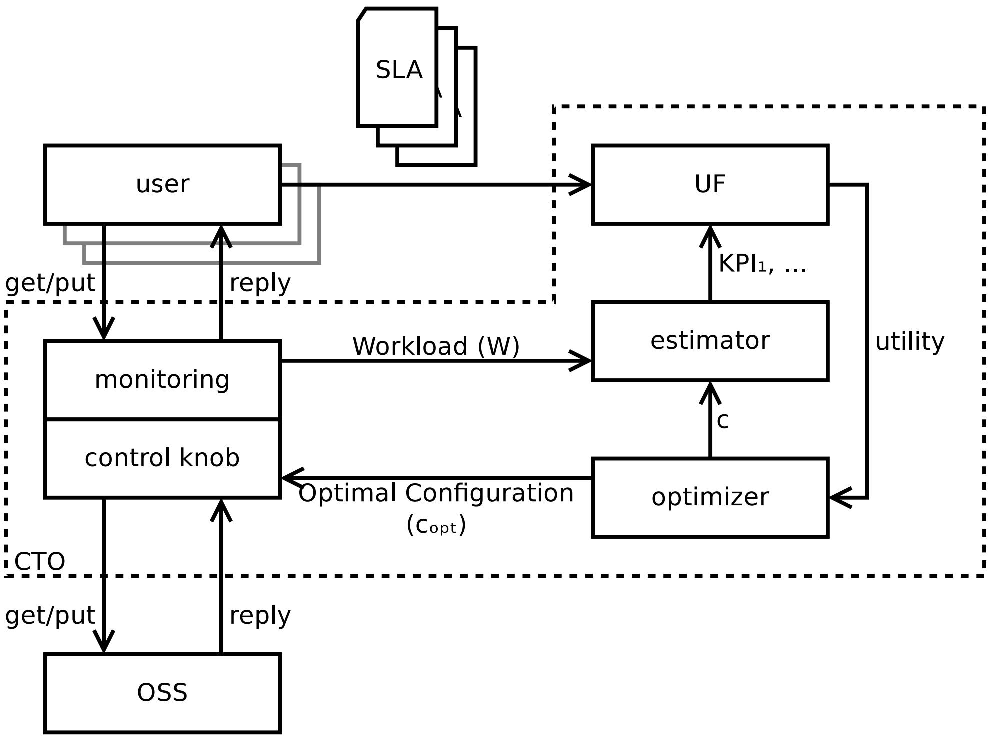

Figure 1: CTO architecture.

User-delay knobs enforce delays on the user-side, by *rejecting* a request and asking the user to retry later. This may be transparently handled inside a client library, so that the user application itself is unaware of the underlying delay mechanism. For improved interaction, the system may transmit along with a rejection notification the time after which the user should retry the request, which we call Re-Think-Time (RTT). User-delay knobs have the advantage that they require less state on the system-side, e.g., no queue nor an open connection needs to be maintained. Moreover, they give more insight about the observed performance: Users may collect statistics about the number of retries and the accumulated amount of RTT, to decide whether an upgrade of SLA is required to improve performance.

In case of CTO, we decided to implement a user-delay knob as follows. The user-side library is configured with a fixed RTT. On the system-side, users are grouped into *classes* of similar SLAs. A *storage configuration c* consists of rejection probabilities for each class and each request type (get or put). With probability 0, no request of a user is rejected, hence she observes the maximum possible performance. As the probability increases, requests are more likely to be rejected and the user is more likely

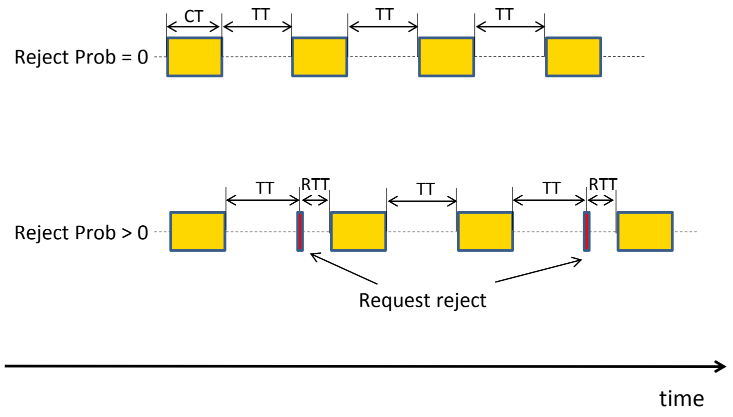

Figure 2: Workload with user-delay control knob. CT is the request completion time given by the performance of the OSS, TT is the user think-time, and RTT is the re-think-time after a request is rejected.

to retry, hence introducing more RTT and lowering the performance observed by the user. Thus, the user-perceived performance is determined by the request processing time of the OSS and the sum of all RTT (Figure 2).

4.3 Utility Function

As required by our problem statement, the optimization goal is expressed to CTO as a Utility Function (UF) defined over the values of the KPIs:

$$utility = UF\left(valueOf(KPI_1), \ldots, valueOf(KPI_m)\right) \tag{1}$$

Ultimately, a UF should reflect profits of the storage provider. These profits may be decomposed into customer payments, penalties associated with SLA violations, customer retention rates due to poor performance, hardware and software amortization costs and other provider-specific aspects. The choice of the UF is up to the provider, however, we give an example of such a function in the evaluation section.

4.4 Estimator

The estimator is the core of our contribution. Its role is to efficiently estimate the KPI values that would be obtained given the workload W and the storage configuration c:

$$\mathcal{E}(W, c) \to (valueOf(KPI_1), \cdots, valueOf(KPI_m)) \tag{2}$$

The workload W is fed from the analysis done by the monitoring component, whereas the configuration c is left as a free variable to be chosen by the optimizer.

Before explaining how the estimator works, let us define the *number of concurrent readers* n_r as the number of users who are waiting for a read request to complete, i.e., they are in the CT period as illustrated in Figure 2. Note that, n_r is smaller or equal to the number of users reading, as some of them may be thinking. Similarly, we define the *number of concurrent writers* n_w.

To output accurate values, the estimator needs to understand the raw performance of the OSS. A calibration table is used to map a number of concurrent readers and writers to measured KPI values. As an example, for $n_r = 5$ and $n_w = 5$, the OSS may obtain a total read throughput of 100 MB/s and a total write throughput of 50 MB/s. Calibration may be done either offline or online. In the online case, the calibration table is constructed on-the-fly based on measured values of n_r, n_w and KPIs, while the system is in production. For simplicity, we opted to use offline calibration and build calibration tables as follows: CTO is set to calibrate mode to override rejection probabilities to zero, effectively eliminating RTT. Then, a workload generator simulates concurrent users requesting random objects from the OSS, with zero think-time, so as to iterate over a wide range of concurrent readers and writers. The obtained KPI values are stored in a file and later used by the estimator.

Let us now explain how to estimate KPI values given the calibration table, the workload analysis W and the storage configuration c. First, for each class k and each request type t (get / put), the expected re-think-time $RTT_{k,t}$ is computed. Note that, a request may be rejected several times, hence the expected value resembles a power series. Next, for each class and request type, we aim to compute the probability of having n concurrent requests, denoted $p_{n,k,t}$ is:

$$p_{n,k,t} = \binom{N_{k,t}}{n} \frac{(CT_t)^n + (TT_{k,t} + RTT_{k,t})^{N_{k,t}-n}}{(CT_t + TT_{k,t} + RTT_{k,t})^{N_{k,t}}} \tag{3}$$

where $N_{k,t}$ is the total number of users in class k with request type t, CT_t is the expected request completion time of the OSS and $TT_{k,t}$ is the average think-time. $N_{k,t}$ and $TT_{k,t}$ are stored inside the workload W, while CT_t are to be determined as explained below. Once we have the values of $p_{n,k,t}$, a weighted sum with probabilities and values from the calibration table can be used to estimate the KPI values observed by each request type.

However, as highlighted in Equation 3, computing the probabilities $p_{n,k,t}$ requires knowledge about the expected request completion times of the OSS, both for read and write requests –CT_r and CT_w, respectively. Intuitively, this is due to the fact that we need to know the ratio between the duration a user spends thinking (or re-thinking) and the duration she waits for a request (see Figure 2). CT_r and CT_w in turn, depend on the probabilities $p_{n,k,t}$. To break this dependency cycle, we adopted an iterative approach: Some initial values are assumed for CT_r and CT_w, then $p_{n,k,t}$, CT_r and CT_w are computed until convergence.

õn summary, a calibration table of raw OSS performance is used to estimate the KPI values for each request type.

4.5 Optimizer

The optimizer uses the estimator function to find the storage configuration c_{opt} that minimizes the UF:

$$c_{opt} = \min_{c} UF(\mathcal{E}(W, c)) \tag{4}$$

This is essentially a global optimization problem over all the possible storage configurations. Given the large search space and the possibility of local minima existence, we chose to implement the optimizer using simulated annealing [13].

5 Evaluation

In this section we evaluate CTO. First, we describe the platform on which experiments were done and the workload model. Then, we present and analyse the obtained results.

5.1 Platform

To validate CTO and to show that it manages to maintain SLAs in a platform-agnostic manner, we chose to run experiments over three scenarios, featuring increasingly complex storage topologies: a desktop, a single server and a storage cluster. First, we ran experiments on a desktop featuring commodity hardware and a simple storage topology, i.e., a single HDD at 7200 rpm. The measured throughput of the disk is approximately 65 MB/s for sequential access and 55 MB/s for random access, as measured using the IOzone [3] benchmark.

Second, we ran experiments on a single high-end server, featuring four HDDs at 7200 rpm configured in a RAID10 array. To make sure the performance is disk-bound, we limited the amount of memory to 1 GB. The measured throughput of the array is approximately 250 MB/s for sequential access and 57 MB/s for random access. Note that the random access performance is highly unpredictable, as the RAID controller schedules requests over the four disks without informing the software about its decisions.

Finally, we performed experiments over a storage cluster build using the Ceph distributed filesystem [32]. We used a total of 4 servers with the same memory and disk configuration as the one in the single-server scenario. One server was hosting the Ceph client, Ceph meta-data server (MDS) and CTO. The other three servers were used for the Ceph object-store daemon (OSD). The machines were linked using a high-end, non-blocking Gigabit Ethernet switch. Note that Gigabit Ethernet can transfer approximately 100 MB/s (after deducing protocol overheads), which means that for sequential access the network is the performance bottleneck and not the disks themselves. In contrast, for random access, the disks are the bottleneck. This highlights the complexity of this scenario, which is specially designed to test CTO in an extreme case.

5.2 Workload

We here describe a workload model based on published statistics on storage workloads and video-on-demand websites in particular. Before running the tests, we initialized the OSS to store 1000 files. The file sizes are generated using a Pareto distribution (minimum 2 MB, $\alpha = 1$) as found in [6].

The storage system is accessed concurrently by a number of users, n_R of which are doing read requests and n_W are doing write requests. Each user works in a closed-loop [8, 25] as follows: The user starts by requesting a random file from the OSS, waits for the request to complete, waits for a random think-time and repeats. For deciding which file to request next, we sample the Zipf distribution which is a good model for file popularity on the Internet [5]. With respect to the think-time, we use the exponential distribution as done in the TPC-W industry standard for web benchmarking [22].

5.3 CTO Configuration

Each user is randomly assigned to one of the three SLA classes, gold, silver or bronze, configured with a target throughput of 12 MB/s, 5 MB/s and 2 MB/s, respectively. Each class is associated with a piecewise constant (or step) penalty function, that captures revenue losses in case the OSS is unable to deliver the promised performance. We have chosen this particular penalty form as it is less sensitive to measurement noise and more intuitive from the user's perspective compared to linear penalty function [12]. The UF used by the provider is the sum of the penalties. Hence, CTO's task is to minimize the sum of penalties paid.

5.4 Analysis

In this section, we present and analyse results obtained on the Ceph storage cluster. This is due to the fact that Ceph is a more realistic representation of cloud storage systems. Hence it shows CTO's behaviour in its intended environment. Note that, experiments on other platforms gave better results.

We performed 130 experiments with various workloads. Note that, due to think times, the average number of concurrent requests directed at the OSS is lower than the number of concurrent users. Each workload generated a certain average number of concurrent read and write requests at OSS. We measured workload intensity as a sum of the number of concurrent read and write requests at OSS. The lowest workload generated 0.513 concurrent requests, while the highest 148.83. For 124 workloads SLAs were violated resulting in positive UF (sum of penalties).

Since our UF reflects the sum of penalties, our criterion for CTO success was *penalty improvement* – the reduction of UF, measured as the ratio of the UF value with CTO deactivated and the UF value with active CTO. The average penalty improvement was 3.23, its maximum value is 68.43 and minimum value is 0.56. These results show that on average CTO reduces penalties by multiplicative factor 3.23.

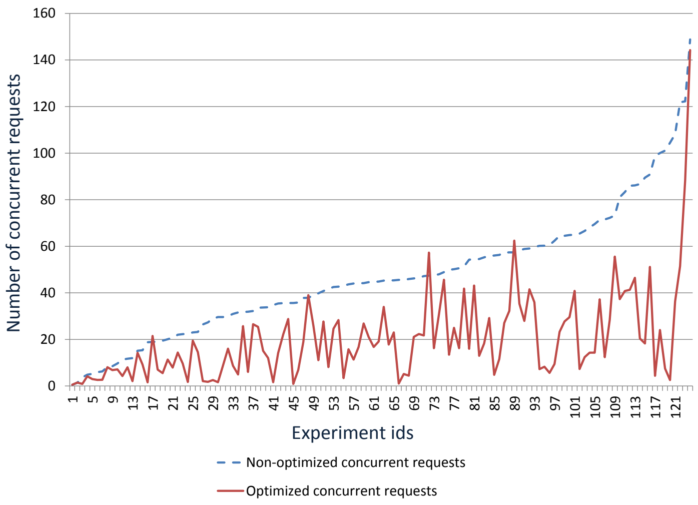

Figure 3: CTO strategy revealed: CTO keeps the number of concurrent requests below a threshold around 40. The x-axis reflects experiment IDs, ordered by increasing number of concurrent requests of non-optimized experiments.

Figure 3 reveals the CTO strategy, which is the result of the optimization procedure in Equation 4. The figure plots the number of concurrent requests at OSS for non-optimized and optimized experiments. The results show that, as the number of concurrent requests in non-optimized experiments increases, CTO strives to keep the number of concurrent requests in optimized experiments below a limit around 40. Since the OSS throughput is shared among all users, when the number of concurrent requests grows, the OSS throughput approaches saturation. At this point, throughput per user drops and, for non-optimized experiments, gold SLAs start to incur high penalties. CTO avoids this situation by increasing the rejection probabilities for silver and bronze requests, while keeping the overall number of concurrent requests at OSS below some threshold.

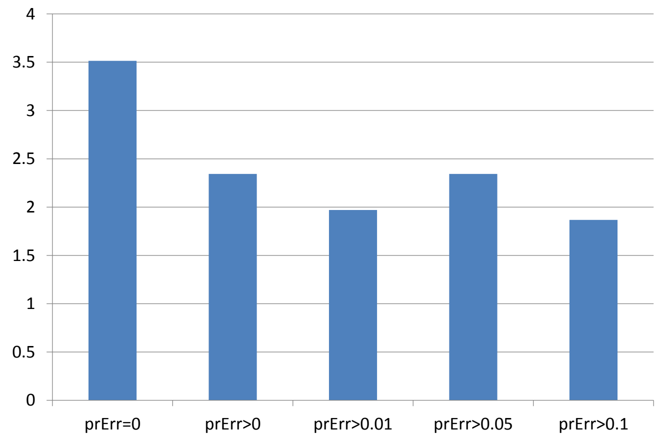

Figure 4: Influence of error probability.

Our calibration on the Ceph distributed filesystem was limited to 20 concurrent readers and 20 concurrent writers. As Figure 3 shows, the number of concurrent requests in our experiments exceeded this calibration numbers. Thus, some probabilities in Equation 3, specifically those for high values of n, were not estimated. The sum of these probabilities is the *error probability* of the estimator. We estimated this error probability and analysed its influence on CTO's effectiveness, i.e., penalty improvement (Figure 4). From these results we conclude that a positive error probability reduces penalty improvement. Consequently bigger calibration tables are desired. However, as demonstrated in Figure 3, CTO strives to keep the number of concurrent requests low, diminishing the contribution of high loads to error probability. Thus, CTO demonstrates stability against high-load analysis.

6 Conclusions

We presented CTO, an OSS-agnostic solution for providing performance guarantees to cloud storage. We thus contribute a key issue for allowing new users to adopt cloud storage. We presented the overall design that was centered around the generic rejection probability control knob and the specifically designed performance estimator. Evaluation on three different platforms showed that considerable improvements in terms of paid penalty may be achieved.

We aim to perform rigorous experimental evaluations and extend the work in several ways such as improving CTO by incorporating on-line calibration and allowing accurate interpolation for missing calibration points.

7 Acknowledgments

This work was partially supported by the EU project VISION Cloud under grant agreement no. 257019.

References

[1] Amazon s3 service level agreement. Available online: `http://aws.amazon.com/s3-sla/`, last checked: 2015-06-12.

[2] Google cloud platform: Cloud storage. Available online: `https://cloud.google.com/products/cloud-storage/`, last accessed: 2014-01-20.

[3] Iozone filesystem benchmark. Available online: `http://www.iozone.org/`, last checked: 2015-01-05-10.

[4] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[5] A. Abhari and M. Soraya. Workload generation for youtube. *Multimedia Tools and Applications*, 46(1):91–118, 2010.

[6] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3), Oct. 2007.

[7] M. Andreolini, E. Casalicchio, M. Colajanni, and M. Mambelli. A cluster-based web system providing differentiated and guaranteed services. *Cluster Computing*, 7(1):7–19, Jan. 2004.

[8] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. *SIGMETRICS Perform. Eval. Rev.*, 24(1):126–137, May 1996.

[9] S. Arunagiri, Y. Kwok, P. J. Teller, R. Portillo, and S. R. Seelam. Fairio: An algorithm for differentiated i/o performance. In *In Computer Architecture and High Performance Computing SBAC-PAD*, pages 88–95. IEEE Computer Society, 2011.

[10] L. N. Bairavasundaram, G. Soundararajan, V. Mathur, K. Voruganti, and S. Kleiman. Italian for beginners: The next steps for slo-based management. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[11] M. W. Benjamin, C. Bodley, A. C. Emerson, and M. Watts. A saga of smart storage devices: An overview of object storage. *login*, 39(1), Feb. 2013.

[12] C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh. Cooperative negotiation in autonomic systems using incremental utility elicitation. *In Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 89–97, 2003.

[13] P. Brucker. *Scheduling Algorithms*. Springer, 5th edition, March 2007.

[14] G. Fankhauser, D. Schweikert, and B. Plattner. Service level agreement trading for the differentiated services architecture, 1999.

[15] S. V. Gogouvitis, M. C. Jaeger, H. Kolodner, D. Kyriazis, F. Longo, M. Lorenz, A. Messina, M. Montagnuolo, E. Salant, and F. Tusa. Vision cloud: A cloud storage solution supporting modern media production. *SMPTE Motion Image Journal*, 122(7):30–37, Oct. 2013.

[16] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[17] A. Gulati, A. Merchant, and P. Varman. D-clock: Distributed QoS in Heterogeneous Resource Environments. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 330–331, New York, NY, USA, 2007. ACM.

[18] Q. Huang, H.-M. Chen, K.-T. Ko, S. Chan, and K. S. Chan. Call admission control for 3g cdma networks with differentiated qos. In *Proceedings of the Second*

International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications, NETWORKING '02, pages 636–647, London, UK, UK, 2002. Springer-Verlag.

[19] K. Jackson. *OpenStack Cloud Computing Cookbook*. Packt Publishing, Sept. 2012.

[20] E. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Performance-based service differentiation in clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 505–514, May 2015.

[21] Y.-D. Lin, C.-M. Tien, S.-C. Tsao, R.-H. Feng, and Y.-C. Lai. Multiple-resource request scheduling for differentiated qos at website gateway. *Comput. Commun.*, 31(10):1993–2004, June 2008.

[22] D. Menasce. Tpc-w: a benchmark for e-commerce. *Internet Computing, IEEE*, 6(3):83–87, 2002.

[23] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[24] N. Saxena, K. Basu, S. Das, and C. Pinotti. A new service classification strategy in hybrid scheduling to support differentiated qos in wireless data networks. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 389 – 396, june 2005.

[25] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[26] P. Skowron, M. T. Biskup, L. Heldt, and C. Dubnicki. Fuzzy adaptive control for heterogeneous tasks in high-performance storage systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 13:1–13:11, New York, NY, USA, 2013. ACM.

[27] V. Sundaram and P. J. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In K. Jeffay, I. Stoica, and K. Wehrle, editors, *IWQoS*, volume 2707 of *Lecture Notes in Computer Science*, pages 479–497. Springer, 2003.

[28] A. Undheim, A. Chilwan, and P. Heegaard. Differentiated availability in cloud computing SLAs. In *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, pages 129–136, 2011.

[29] S. Uttamchandani, K. Voruganti, S. M. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage qos management beyond a "4-year old kid". In *FAST*, pages 31–44, 2004.

[30] J. Varia and S. Mathew. Overview of amazon web services. Technical report, Amazon, Jan. 2014.

[31] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[32] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[33] R. Wijayaratne and A. L. N. Reddy. Providing qos guarantees for disk i/o. *Multimedia Syst.*, 8(1):57–68, Jan. 2000.

[34] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *International Workshop on Quality of Service*, pages 75–91. Springer-Verlag, 2001.

[35] L. Wu, S. K. Garg, and R. Buyya. SLA-Based Resource Allocation for Software As a Service Provider (SaaS) in Cloud Computing Environments. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 195–204, Washington, DC, USA, 2011. IEEE Computer Society.