Improving OpenStack Swift interaction with the I/O Stack to enable Software Defined Storage

Ramon Nou, Alberto Miranda, Marc Siquier
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{ramon.nou, alberto.miranda, marc.siquier} @bsc.es

Toni Cortes
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya
toni.cortes@bsc.es

Abstract—**This paper analyses how OpenStack Swift, a distributed object storage service for a globally used middleware, interacts with the I/O subsystem through the Operating System. This interaction, which seems organised and clean on the middleware side, becomes disordered on the device side when using mechanical disk drives, due to the way threads are used internally to request data. We will show that only modifying the Swift threading model we achieve an 18% mean improvement in performance with objects larger than $512\,$KiB and obtain a similar performance with smaller objects. Compared to the original scenario, the performance obtained on both scenarios is obtained in a fair way: the bandwidth is shared equally between concurrently accessed objects. Moreover, this threading model allows us to apply techniques for Software Defined Storage (SDS). We show an implementation of a Bandwidth Differentiation technique that can control each data stream and that guarantees a high utilization of the device.**

Index Terms—**SDS;OpenStack Swift; Kernel Interaction;Storage; QoS; Fairness**

I. Introduction

OpenStack [1] is a world-wide used middleware that offers an Amazon Web Services-like[1] service in an open source way. Such services span across different layers: from the computing layer, where we can find services such as Nova that are able to create and run virtual machines, to the storage layer, where we can find major storage services like *Cinder* for Block Storage, *Glance* for Image Storage, and *Swift* for Object Storage. In this paper we analyse *Swift*, the Object Storage Service.

A general problem frequently found in middlewares is that they often rely on a pool of threads for dispatching I/O requests to the filesystem, and these requests are not optimized with regards to the needs of either the operating system or the underlying hardware. In the case of *Swift*, as we will observe in the analysis section, read operations are dispatched by using a round-robin strategy among different threads. Unfortunately, a pool of threads dispatching parts of an object/file

uncoordinated, as it happens in the case of the round-robin policy of *Swift*, introduces entropy and noise inside the I/O scheduler. For example, I/O schedulers such as Completely Fair Queuing (CFQ) [2] or the ones found at Split-level [3], include heuristics and different queues to optimize requests produced from different Thread IDs (TIDs) assuming that requests to the same file will most probably come from the same thread. The behaviour of *Swift* clashes with these optimizations, given that requests to the same object are separated in the operating system per TIDs. This causes that, even if the objects accessed by the different threads are the same, the OS perceives them as requests from different applications.

More concretely, in *Swift* requests from different objects are served in a $64\,$KiB chunk basis (*Swift* chunk size) that are distributed among the available threads, thus the order in which the I/O scheduler sends requests to the disk is not optimal due to the loss of the I/O context. This behaviour has a big impact with rotational devices (i.e. HDDs) as the I/O scheduler is not working as it was intended. With respect to non-mechanical devices (i.e. SSD or Non-Volatile Memory (NVM)), given that schedulers used with such devices do not take I/O context into account, we see no clear penalty of this behaviour, though this could change if new context-based schedulers appear for these devices. Summarizing, in general-purpose devices, whose schedulers are optimized to get more performance from contiguous requests, this produces significantly **degraded performance**. Additionally, since the I/O scheduler is not able to distinguish requests coming from different *Swift* clients, enforcing a fair distribution becomes problematic: if the workload of a *Swift* client provokes more worker threads to be assigned to it, the I/O scheduler will try to serve these requests to optimize seek time[2], which may produce **starvation** to the other clients.

This paper proposes a change in how *Swift* is used in order to distribute requests so that the I/O scheduler

---

[1]aws.amazon.com

[2]Since it will consider them to be I/O related.

does not lose the semantic context, thus allowing it to optimize requests to reduce latency. In addition, we use this mechanism to implement a bandwidth differentiation that will guarantee that users or requests achieve the agreed performance regardless of the other clients in the system. We focus our evaluation on HDDs given that there are no standard I/O schedulers specialized for SSDs that may benefit from our proposal.

The remaining of this paper is organized as follows: we present a description of *Swift* and the I/O Scheduler in Section II, followed by a deep analysis of OpenStack *Swift* (Section III). We introduce a proposal to solve the issues detected (Section IV) and possible implementation of a SDS bandwidth differentiation mechanism (Section V). We finalize with the evaluation (Section VI) and conclusions.

II. Component description

The two components that come into play when a user asks for an object in Swift are the object store in *Swift* and the I/O Scheduler in the kernel.

A. OpenStack Swift - Object Store

Swift architecture builds up over a set of object storage servers and one or more proxy servers. *Swift* offers to the user a REST API to manage objects (GET and PUT), and manages other storage aspects such as replicas or erasure codes. The user sends a GET request to the *Swift* proxy and the proxy sends it to any of the Object Storage (**OS**) nodes containing the object. Once the object server has the request, a *python* iterator is returned to the user. The iterator offers to the user chunks of data, issuing read operations at the Object server.

Swift is configured at boot time with two main parameters: the number of workers, that serve each TCP request, and the number of I/O threads per worker. The workers are created in the Web Server Gateway Interface (WSGI) [4] layer, and each data chunk for a request is served inside the object server by using one of the configured *n* threads. Thus, the separation between worker and thread can be considered as a two level division: first, all threads inside a worker are served in round-robin, and when exhausted, the next worker is served (a diagram can be found at the top of Figure 3). Nevertheless, as we discuss in Section III, choosing these parameters on dynamic environments for each individual object server is difficult, since the optimal parameters will depend on the expected load.

This paper analyses the behaviour of (i) two configuration parameters, and (ii) the I/O request behaviour inside **each single object server**. The solution proposed is intended to get improved I/O for each of the available object servers, and to enable the capabilities of doing more complex controls like a distributed bandwidth enforcement. For this reason, the analysis and evaluation is done inside a single object server. Nevertheless, a complete *Swift* evaluation for the bandwidth differentiation with interferences is presented in Section VI-B2, to demonstrate how a simple distributed mechanism for bandwidth enforcement can be implemented using the low-layer mechanism presented.

B. I/O Scheduler: CFQ

The I/O Scheduler is a component in the Linux kernel (and other operating systems as well), that reschedules, merges and transforms all the I/O requests going to a **single device**. It is really important for mechanical devices as it is aimed to reduce seek time and enforce fair sharing of the device. CFQ [2] is one of the available I/O scheduler in the Linux Kernel. It is considered one of the most advanced as it can separate requests from different processes, and even consider different processes as close collaborators and schedule their requests close to each other to reduce seek time. As mentioned, this close collaborator hint produces performance and fairness problems when we use a round-robin thread pool to send I/O requests to the storage device. CFQ is the only I/O scheduler built to maintain the fairness between different processes, something that is desired when we have a middleware that request different objects and want a fair sharing of resources. It also uses is the I/O priority field, so it can prioritise different request giving the capability to create different policies.

III. Analysis

For the analysis of the original *Swift* behaviour we used a single device (ST91000640NS), the reason being that the I/O scheduler actions which we want to study and optimize, are applied to a single device even on a RAID or a large cluster. In order to generate the traces for the analysis, we use blktrace [5] and Paraver [6] using a translator tool called blktrace2paraver [7]. Such tools enable us to understand what happens inside the system (i.e. reads and writes) with great detail.

For the analysis, we exercise the setup with two different workloads. For the first workload, we store 1 GiB objects with random content and each client asks for a different object to avoid optimizations of the I/O stack and cache hits. For the second workload, we use smaller objects of 64 KiB (a smaller I/O chunk in *Swift*) to evaluate our proposal in a scenario serving small objects (see Section VI).

We evaluate concurrent requests to different objects with different *Swift* worker/thread configurations to observe their effect on performance and the disk. We show a summary of the results in Figure 1 (Performance) and in Figure 2 (Fairness) requesting 2, 4, 8, 16, and 32 big (1 GiB) objects.

Figure 1 shows the distribution of the bandwidth obtained **per object** using different parameters of workers and threads (1, 2, 4, 8, 16 and 32 threads and 1, 2, 4, 8, 16 workers while $threads * workers \leq \#objects\ requested$).
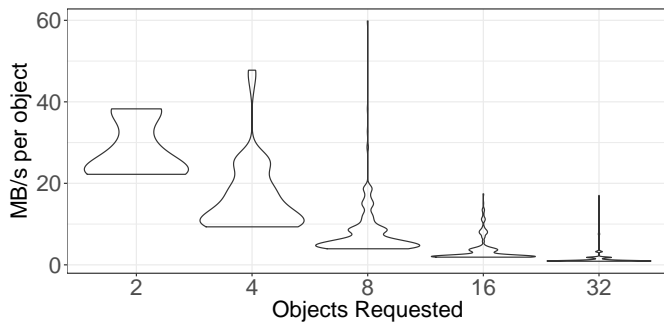
Figure 1. Performance obtained using the original *Swift* with different workers and threads numbers and with requests for 2, 4, 8, 16 and 32 big (1 GiB) objects. Each violin plot shows the distribution of the Bandwidth obtained per object. Wider parts of the violin has more occurrences than the other parts.
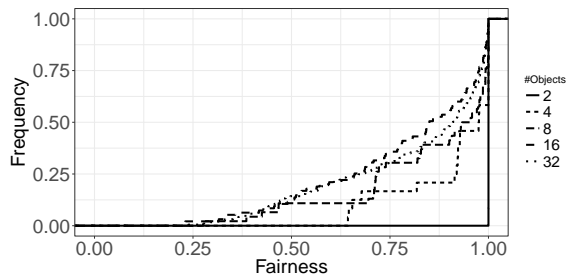

Figure 2. Fairness obtained using the original *Swift* with different workers and threads numbers and with requests for 2, 4, 8, 16 and 32 big (1 GiB) objects. The ECDF shows the distribution of the fairness obtained, being 1.0 the fairest. i.e., the point (y = 0.5, x = 0.9) with 4 objects means that the 50% of the observations have a fairness below 0.9.

The distribution is plotted using a violin plot. We can spot how in some cases the obtained range is very large. To complement this figure, we can see in Figure 2 the distribution using an empirical cumulative distribution function (ECDF) of the fairness[3] obtained for each tested configuration. The *ECDF* shows how the values below the x-axis point are distributed on the y-axis point value (i.e., for 32 objects, a 25% of the experiments present a fairness below 0.75). As we anticipated from Figure 1, fairness is not maintained on a large number of configurations and it decreases when the number of objects increases.

Analysing the non-aggregated results (looking at the individual worker and thread combinations from Figure 1 and Figure 2), we get worse performance in fair situations if the number of workers is lower than the number of simultaneous objects requested. Increasing the number of threads does not help, since the workers parameter is shaping the performance values more than the number of threads. On the other hand, an interesting situation is observed using 1, 2 and 3 workers with

---

[3]Fairness is calculated using *Jain's fairness index* [8] $\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$ where x_i is each object's bandwidth.

4 simultaneous objects: with 1 worker we get a fair allocation but bad performance for each object, but with 2 workers the scenario is totally unfair: we obtain more than 50% of the performance for one single object and the rest for the three other objects. As requests share the last worker when they are busy, one worker is serving only one object and the other worker is serving with round-robin the remaining three objects producing a large number of seeks. The same behaviour is observed when we have less workers than objects requested: increasing the number of threads does not show any improvement and in fact, it decreases the obtained bandwidth. We can observe that even if the bandwidth distribution is unfair, the performance is better as we burst requests and we reduce the seek time.

Therefore, our objective is to eliminate these configuration issues by using a dynamic threading model that allows to **reduce the unfair configurations** as they introduce unexpected performance changes, and to **reduce/simplify the configuration settings**, as we have seen the selection of the right combination of those parameters is hard as it depends on the expected workload, it can not be changed without restarting each object server, and it should be set up on each object server. Improving the performance is not the main objective because performance in unfair configurations will be normally better as the bursts are longer and the seek time is reduced.

A. Detailed analysis

In this subsection we discuss some of the issues observed when analysing several representative configurations and scenarios in our experimental setup.

a) 1 Client, 1 worker, 1 thread: We observed that *Swift's I/O thread* only submits one request each time, but the operating system issues additional read requests due to prefetching. **Using one thread to serve one object, reduces latency as the operating system is issuing sequential prefetch requests.**

b) 1 Client, 1 worker, 4 threads: If more threads are available than objects requested, requests are distributed among 2 of the 4 threads available as we do not have sufficient work for all of them. Due to this, we observed that the disk also had a maximum of 2 operations in-fly, but the operating system's prefetching did not start given that operations come from different threads. With this type of distribution, we are effectively creating **false cooperative threads inside the I/O scheduler (CFQ), as we mistakenly assume that a single thread will serve the same object**. This can also affect other applications, since the OS cannot do anything to help with fairness if it does not understand the mapping between I/O requests, objects and threads. In this particular scenario, the bandwidth obtained for such a request will be 66% higher than other I/O request using a single thread at the same time. This behaviour damages fairness as we

are offering more I/O time to this request than the other ones.

c) 2 Clients, 1 worker, 1 thread: In this case, we add a new client request, while we maintain 1 thread. By analysing which object the thread served, we observed that the thread changed the object to be read from intermittently as the worker served them using round-robin. **This behaviour is totally fair, but we are losing performance with mechanical devices due to seeks, since a first I/O scheduling is done in the middleware level**. It is better to do this sharing at the kernel level, as it has knowledge of all the requests going to the disk.

d) 2 Clients, 1 worker, 4 threads: With 4 threads, each thread is serving requests but each object is requested by all four threads. **The object swapping inhibits opportunities for prefetching and optimisations on the kernel.** As a result the performance drops from 75 MB/s to 45 MB/s compared to the best case. Specialising threads, so that the kernel I/O scheduler can generate a better schedule and avoid serving other streams from the same thread, will increase the performance.

e) 2 Clients, 2 workers, 1 threads: With 2 workers and 1 thread, we observed that each request goes to a different worker. The disk is shared in a coarser way (due to the first scheduling point shown in Figure 3) than with 1 worker and each thread serves a single object, hence the performance on the disk is better as it implies less seeking (requests are easily merged and batched). The behaviour for each of the objects is the same as the 1 Client, 1 worker, 1 Thread trace: Kernel prefetch is also working for both objects. **The device is better used. This is due to the larger number of requests to the same object done by each worker.** Creating bursts of sequential requests increases the performance obtained. As each worker has the same number of requests, the performance of the two objects is fair, however, we still control the scheduling of requests in the middleware layer, so the resources may not be used correctly.

f) 4 Clients, 1 worker, 4 threads: The fairness problem shows up when we have requests for different objects going into *Swift* and each request, even from the same object, is served by different threads. In this scenario, the 4 threads are issuing requests in round-robin, so there is no fixed mapping. The I/O scheduler, gives more I/O time to a specific object due to the close cooperator mechanism in CFQ, and produces starvation on other objects. Using other schedulers may avoid this particular problem, but they can not use I/O priorities and are not created for having fairness between requests so our capabilities and OS interaction from the middleware is reduced. The discussed results show that serving a single object from the same thread increases the performance of the system. For this reason, we propose a new threading model that assigns one thread to a single object and therefore removes the need to setup the workers and threads parameters. Using this method we may have
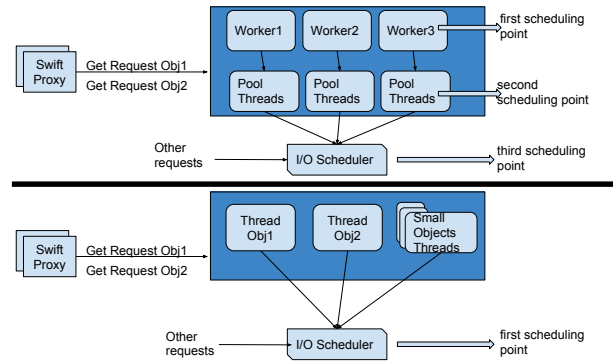

Figure 3. Original Swift threading model (top) compared to the modified Swift threading model (bottom)

contention if requests go to the same object, but only for 64 KiB blocks which is the disk chunk served by *Swift*. Such contention, due to the small size, does not have a major impact.

IV. NEW THREADING MODEL IMPLEMENTATION

From the analysis discussed, we observed that requests are served in round-robin by all the threads available in *Swift*, which often interferes with the optimizations of the kernel's I/O scheduler. To solve this and produce a good I/O scheduler behaviour we need to fix the thread with the stream, effectively mapping a TID with an Object ID. By doing that, **we can eliminate the workers configuration value** while obtaining the same effect, and the CFQ scheduler would work as intended as requests will follow the rule *one stream - one TID*. CFQ is the only Linux default I/O scheduler that uses such behaviour, but more can be found in the literature.

As we can not know the number of objects in advance, we provide a dynamic creation of threads so **we can remove the threads parameter** from the configuration. With this dynamic creation we serve each object with one thread, it removes the explicit I/O wait and the two aforementioned configuration options from *Swift*. On the one hand, parallelism (but not performance) is reduced if we have requests going to the same object, however, it will only affect in 64 KiB blocks which is the Swift chunk size as we mentioned before. On the other hand, it will be in general better than a fixed thread configuration as we does not know how many objects we will serve simultaneously.

Our implementation relies on the Object ID to assign work to a thread, while requests to the same object share the same thread. No major changes are done to the code other than changing the *ThreadPool class*. In the case of smaller objects of less than 10 chunks, we redirect them to a populated pool of threads to reduce the overhead, in order to obtain a similar performance while maintaining the fairness. We also introduce a maximum number of threads, to avoid overheads if a larger number of objects

is going to be requested, though the limiting factor will be the storage system. Figure 3 shows a diagram explaining the two threading models: while with the original *Swift* we have three I/O scheduling points (two of them outside the operating system), with our implementation the I/O scheduling is left to the kernel, thus reducing interferences.

Finally, it may happen that an object will be shared among several clients. In this scenario, the assigned thread will issue non-sequential requests, but the I/O scheduler will have the opportunity for merging, prefetching and caching requests on most situations. Of course, randomized access by different clients will harm the performance on both models, but in the case of our threading model, it will not harm the other object requests as all the clients for the same object will get the same I/O thread.

V. BANDWIDTH DIFFERENTIATION

Bandwidth differentiation is the ability of the object store to apply unfair but controlled bandwidth per object. Thanks to the new threading model we can track each stream of data at the object server, which was not possible on the original model and it is an essential service for Software Defined Storage.

In order to implement bandwidth differentiation, we can apply several policies. For instance, we can directly apply I/O priorities [9] to each object request to create bandwidth differentiation policies, controlling the bandwidth or throughput offered to one object, group of objects, or tenant by each individual object server. Having control at the request level and using the OS mechanisms allows sharing spare disk bandwidth, a degree of control that would not be possible at higher layers, as we would not be able to prioritise our requests over other I/O threads due to interferences.

One way to get bandwidth differentiation is to increase the priority of a request if the object is being served below the *needed BW*, and mark it as a low priority request if they exceed the *needed BW* value. The Operating System offers several mechanisms to classify I/O requests, one of such being *I/O Priority*. This mechanism offers 3 request classes on the latest Linux: *Idle*, *Best Effort* (BE), and *Real Time* (RT). BE has 8 priority levels (0 maximum, 7 minimum).

Though *I/O Priority* is rarely used, it allows differentiating between important and non-important requests, which is why we chose it for our threading model. We use only BE level 0 (*BE(0)*) and *IDLE* priorities, so the spare bandwidth is distributed to the other I/O processes. For instance, requests that need to increase their bandwidth should have a higher priority than the default one (e.g. *BE(4)*).

The kernel's I/O priorities mechanism simplifies the implementation of bandwidth differentiation as we can avoid using delays in the code. However, as *Swift*'s requests can not be cancelled and the HDD performance is non-linear, it is a best effort: if the requested bandwidth is not obtained, the client will need to cancel the ongoing transfer, and repeat it. The proxy server will intercept it, and send it to another object storage (selected using some defined policies) with enough capacity.

Distributed and coordinated bandwidth differentiation can be applied using an external controller. However, the actions that the controller can take are limited to modify the bandwidth allocated up or down to specialise storage servers and to reduce seek time. So again we could only achieve a best effort bandwidth differentiation.

To provide such distributed bandwidth differentiation, the IOSTACK project has an SDS controller with agents that provides a control plane to adjust the bandwidth and do actions like distribute and specialize object servers (code available at GitHub [10]). The deep evaluation of the distributed bandwidth differentiation control is out of the scope of this paper.

VI. EVALUATION

In this section we will evaluate the new threading model and then the bandwidth differentiation functionality added into *Swift*.

The first set of experiments (*New Threading Model*) is done with a *Swift* All-in-One installation, using a 7200 rpm HDD for object storage. The workloads are sent to *Swift* using the same machine with the `--no-download` option so it only does the GET call, but does not write anything to the disk avoiding bottlenecks (but the read on the object storage is done). The second set of experiments, evaluates how effective we are at controlling the bandwidth offered to each object and whether performance is improved. We use a single object store within a normal *Swift* installation, however we also include results with 3 object servers for the interferences use case.

A. New Threading Model

We use the same experiments as in Figures 1 and 2 but this time with our threading model. Note that we do not need to configure any parameter (workers or threads). All the scenarios (Figure 4) show a smaller variation on the bandwidth obtained per object (1 MB/s as maximum). Calculating the fairness we get a 1 value for each experiment so we have a **fair** sharing of resources of each object compared to the different configurations done at the experiments over the original *Swift*. Each object is served by its own thread, therefore creating a more friendly I/O Scheduler behaviour and producing a fair distribution of the disk resources. Although it may seem that an excessive number of threads may be created, we should remember that each thread is a request from an

Number of Clients	Improvement
2	37.7 %
4	5.9%
8	15.9 %
16	13.2 %
32	26.7 %

object in that Object Server that goes to a storage device which will be, in the end, the limiting factor.

We are evaluating only read requests as write requests are unaffected, since they are buffered in the kernel immediately. Finally, as the I/O Scheduler affects one device, using a RAID will not harm the model, as each local I/O scheduler will still see sequential requests. For that reason we are doing the evaluation in a single disk.
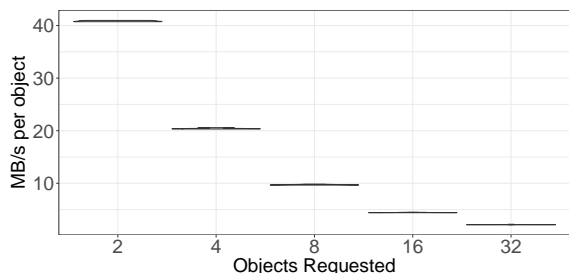

Figure 4. Distribution of the Performance obtained using the dynamic *Swift* per object (1 GiB). The results are more stable than the default scenario.

However, this model is not the best for small objects (i.e., 64 KiB) as they will be served in one chunk, so the overhead of creating a thread creates a performance penalty. Hence, in this scenario we redirect requests to a pool of threads as we explained in Section IV. The original *Swift* results obtain more performance in the 256 and 512 simultaneous objects scenarios but our threading model gets more fairness. The fairness obtained with our modification is higher than 0.85 for more than the 75% of the experiments on the 512 objects case (compared to a fairness below 0.5 for the 75% of the experiments in the original *Swift*). For the other scenarios we achieve similar values: for example with 16 small objects, for the original threading model we observe a 75% of the experiments with a fairness higher than 0.75, whereas with the modified threading model all experiments show a fairness of 1.0. Getting more performance is difficult given that serving small objects in a fair way increases seek time. On the other hand, our results are more stable and predictable. On the performance side with bigger objects, we have a performance gain going from 5.9% to a 37.7% which is produced by a better I/O Scheduling in the kernel. We can see the results in the Table I.

With all these modifications, we have performance improvements with HDD devices, as the kernel's I/O scheduler can do a better work. We have also checked the effect of these modifications with SSD devices, but as there is not a particular context-aware I/O scheduler for SSDs, there are no benefits in fairness due to the lack of penalties due to seeks. In fact, performance is 4% worse (maximum) as dispatching requests as they come will typically offer better parallelism. To circumvent this issue, we can detect if the target device is a non-mechanical device and use a pool of threads to send more requests to the SSDs.

Summarising, although we have performance improvements serving big objects, the most important benefits are: 1) the **removal of the two configuration parameters** (workers and threads) simplifying the deployment and the result of 2) an **improved fairness** in all the scenarios. Smaller objects obtain a good fairness value (over 0.90), while the performance is similar.

B. Bandwidth Differentiation

Bandwidth differentiation can be explained as the creation of a **controlled unfair sharing** of the resource.

1) Experiment 1. Mean bandwidth obtained: To test bandwidth differentiation we use CosBench [11] as a benchmark. This benchmark imitates several (distributed) clients requesting objects from *Swift*. As we do not have any cancel operation on the server side to provide a way to relocate requests, the bandwidth allocation tries to maintain a fair relation with the allocated bandwidth. We will also observe that having bandwidth allocations creates some bursts in the data and HDD devices work better than without bandwidth allocation. This is the same effect that causes that some original worker-thread parameter, the unfair ones, obtain more performance than others.

CosBench uses 300 MiB objects, 300 seconds, using 2 drivers and 8 workers per tenant. Tests are done with 3 tenants. We do not use smaller objects as the performance obtained from CosBench only takes into account instant bandwidth, so when it is not requesting the metric is zero and it does not produce stable results. Our system can maintain the bandwidth required per tenant, using a configurable window (i.e., 15 seconds by default). A larger window produces more stable results but reacts slower to changes in the workload or interferences.

Table II presents performance numbers for different bandwidth allocations, including no allocation and the original *Swift*. Here we obtain better performance, even without bandwidth allocation, due to a better scheduler behaviour. However, we achieve better disk performance when we offer different bandwidth at each tenant due to a more bursty and a behaviour prone to merge I/O. Observing the ∞ bandwidth line, it is interesting to note that CosBench did not manage to get some objects due

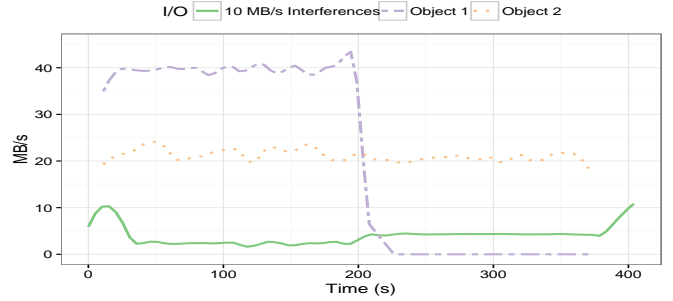| Experiment	Tenant 1	Tenant 2	Tenant 3	Total BW
No BW Diff	8±0.1	8±0.1	7.9±0.1	23.95±0.3
BW: $\infty/-/-$	101.8±0.6	NA	NA	101.80±0.6
BW: $50/-/-$	67.4±4.2	4.8±0.6	4.7±0.5	76.85±5.3
BW: $70/-/-$	78.2±2.5	4.7±1.5	4.7±1.5	87.51±5.5
BW: $25/25/-$	32±5.1	30.4±5.4	3.9±0.7	66.32±11.2
BW: $15/20/15$	16.6±1.7	24.6±4.3	17±2.1	58.13±8.1
Original Swift	7.7±0.3	7.7±0.3	7.7±0.3	23.15±0.9

Figure 5. Performance obtained with 2 tenant requesting two 10 GiB objects with 40 MB/s and 20 MB/s of requested bandwidth differentiation. Background I/O noise (10 MB/s sustained) applied.
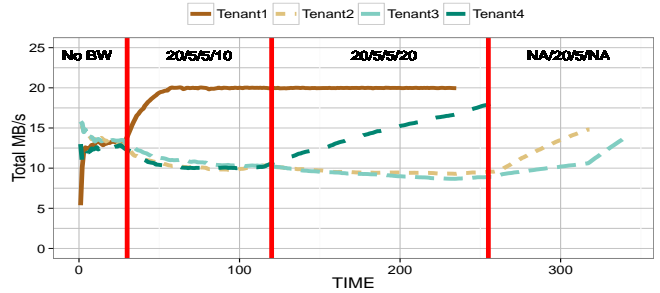

Figure 6. Performance obtained with 4 tenants requesting 5 GiB objects. The requests go to the same object server. We setup the bandwidth as -/-/-/-, 20/5/5/10, 20/5/5/20 and 20/5 values during the timeline. At the last update, as the server is not overloaded, both tenants are increasing their throughput but Tenant 2 has higher priority.

to timeouts on the client side. In this situation, the client should request the object again and the proxy server will move it to another server. Based on these results, it is easy to observe that the concept of "maximum bandwidth" or "maximum throughput" is hard to define on HDDs due to its dependency on the workload. Therefore having control points that use that concept as a metric to distribute objects will produce wrong results as the "maximum" can not be predicted or calculated. For non-mechanical devices, on the other hand, we will not have this penalty since the priorities will distribute the bandwidth accordingly in a fair way.

2) Experiment 2. Bandwidth differentiation with external interferences: We tested the bandwidth differentiation working through time and introducing external interferences. As we cannot cancel requests, we should be proactive and try not to send the objects to an overloaded server. On this experiment we request two 10 GiB objects without bandwidth differentiation with a background interference of 10 MB/s. The figures show the time as $x-axis$ and the obtained bandwidth as the $y-axis$.

With the original *Swift*, requests do not get a lot of throughput due to the background noise that we artificially created in this experiment (but note that it can be naturally produced by the *Swift* replication mechanisms, for example). We get less than 2.5 MB/s for each of the objects and the interference.

If we setup bandwidth differentiation with (40 MB/s and 20 MB/s), requests start to be reordered and prioritised, and burst opportunities start to arise. Figure 5 shows how a request obtains 40 MB/s and the other object 20 MB/s. As observed, the bandwidth obtained does not go higher than the required level, due to the background I/O activity (continuous line) which has normal priority. But the required level is guaranteed thanks to the low-level implementation, that manages all the I/O in the node and issues high priority requests when bandwidth drops from the required level. On the other hand, a positive effect is that the global throughput of the HDD increases compared to the original *Swift*.

We also include results from a distributed execution. We start a process doing 20 MB/s interferences in three object servers and asking 50 MB/s for tenant 1 and 20 MB/s for tenant 2 for their requests (hundreds of objects and different sizes going from 16 MiB to 160 MiB and several clients per tenant). The distributed execution involves network (1 GbE) and a proxy deciding which object server (of the 3 available) to use for each request. The requested bandwidth is achieved by the 75% of the sampled bandwidth points for the two tenants. However, an object server may become overloaded with too many requests, and this is the reason of the points that did not achieve the bandwidth requested.

3) Experiment 3. Changing bandwidth differentiation over time: Finally, in Figure 6 we setup 4 tenants requesting 5 GiB objects with different bandwidth allocations over time. The BW differentiation is deactivated at the beginning, and then a *20/5/5/10* MB/s is selected. At the third phase, we change the settings to *20/5/5/20* MB/s, so the last object obtains more performance. Finally once tenant 1 and 4 finish their requests, we fix a *20/5* MB/s bandwidth values. We can see that since the server is not overloaded, both tenants are increasing their throughput, but Tenant 2 has higher priority. This is a nice effect of the kernel implementation as it shares automatically the spare I/O to different requests. We can see this effect on the second phase, Tenants 2 and 3 get up to 10 MB/s.

VII. Related Work

a) Middleware - Kernel cooperation: Typically the middleware is written with portability in mind, however, sometimes this is not optimal and produces negative effects when we are on overloaded systems. For example, Apache Tomcat had a problem with SSL sessions that produced an undesired overhead in the system. To reduce this effect a communication and collaboration of kernel - middleware is explored by Guitart [12] and it helps to avoid the performance problem. However, such improvements need tools to analyse the whole software stack and the kernel - middleware interactions. The paper by Chuanpeng [13] talks about how to handle concurrent sequential I/O streams in a Virtual Machine setup and explains a similar issue in the VM middleware.

b) Bandwidth Differentiation: Bandwidth differentiation involving networks [14], [15] normally uses queues. However, disk bandwidth is affected by the workload (random, sequential, number of requests) and it is not linear. The work IOFlow by Thereska [16] explores the problem from the I/O perspective controlling the I/O Flow using queues and modifying Samba. Another similar work, Libra from Shue [17] implements bandwidth differentiation using a co-design of the application and the I/O scheduler. Our implementation, using the internal kernel I/O scheduler mechanisms, allows putting a priority on each request. This enforces the bandwidth in the disk, and allows to share the spare bandwidth proportionally and controls interferences from other I/O processes. However, using the kernel I/O scheduler has some drawbacks: Write request priorities are lost, as the control is offered to another process. This can be solved with the Split-Level I/O Scheduling work by Yang [3], but requires major changes on the kernel. More concretely in Swift, there is a middleware layer bandwidth differentiation filter by Gracia [18] enforcing PUT and GET requests, we only enforce GET requests, as controlling the priority on the PUT requests can not be achieved on the stock kernel (as it is done by a specialized thread, kswapd). On the other hand, middleware layer controls, does not work well with scenarios with interferences. The usage of the control plane, with our low-level bandwidth differentiation provided better experience on distributed scenarios with interferences, removing the need to configure the maximum bandwidth of the storage device.

VIII. Conclusions

Our analysis of the I/O behaviour of *Swift* had detected that the number of workers and threads have a big impact on the performance obtained. Using such analysis we have successfully modified *Swift* to use a new threading model with low impact on the code that has very promising results using HDDs. This new threading model makes I/O better in both performance and sharing fairness metrics, as we are working as the operating system expects. This newer model also creates the opportunities to better control the I/O in *Swift* and include functionalities such as bandwidth differentiation, of which we offer a possible implementation. We have observed that changing bandwidth allocations creates bursts in the I/O requests to the disk and increases the performance of rotational disk drives. Some of the challenges that we encountered modifying *Swift* were that Python does not have a mechanism to obtain the Thread ID and that we need to take care of stateless behaviour of *Swift*. This new threading model removes two configuration parameters from the object servers, resulting in an easier deployment and allows to have QoS on *Swift*, and serves objects with a fair allocation of resources between them. The code is available on GitHub [10].

References

[1] OpenStack, "OpenStack," docs.openstack.com.
[2] J. Axboe, "Linux block IO—present and future," in *Ottawa Linux Symp.* Citeseer, 2004, pp. 51–61.
[3] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level I/O scheduling," in *SOSP*, 2015.
[4] Python, "WSGI," www.python.org/dev/peps/pep-3333.
[5] J. Axboe and A. D. Brunelle, "Blktrace User Guide," 2007.
[6] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, pp. 17–31, 1995.
[7] R. Nou, "blktrace to Paraver," github.com/mavy/blktrace-utils.
[8] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984, vol. 38.
[9] "I/O prio," man.org/linux/man-pages/man2/ioprio_set.2.html.
[10] IOSTACK, "IOSTACK Openstack SDS enhancements code," https://github.com/iostackproject/.
[11] Intel, "CosBench benchmark," github.com/intel-cloud/cosbench.
[12] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguade, "Session-based adaptive overload control for secure dynamic web applications," in *ICPP 2005*. IEEE, 2005, pp. 341–349.
[13] C. Li, K. Shen, and A. E. Papathanasiou, "Competitive prefetching for concurrent sequential I/O," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007, pp. 189–202.
[14] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International Conference*, 2010, p. 15.
[15] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," in *(APSIPA ASC), 2012 Asia-Pacific*, pp. 1–8.
[16] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *SOSP'13*.
[17] D. Shue and M. J. Freedman, "From application requests to Virtual IOPs: Provisioned key-value storage with Libra," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.
[18] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom, "Crystal: Software-defined storage for multi-tenant object stores," in *FAST 17*.