Technical Disclosure Commons

Defensive Publications Series

August 21, 2018

FAIR SCHEDULING FOR LOW LATENCY AND HIGH THROUGHPUT STORAGE SYSTEMS

Guillaume Ruty

Jerome Tollet

Aloys Augustin

FAIR SCHEDULING FOR LOW LATENCY AND HIGH THROUGHPUT STORAGE SYSTEMS

AUTHORS:
Guillaume Ruty
Jerome Tollet
Aloys Augustin

ABSTRACT

Techniques are described herein for storage systems to guarantee low latency for small requests while maintaining the system's optimal overall throughput. It batches requests, classifies them, fairly allocates resources to them, and provides a mechanism to expedite the processing of small requests. In today's cloud environments, it is critical that diverse applications run by multiple users can share access to generic storage systems without affecting each other's performance.

DETAILED DESCRIPTION

Storage systems are most often deployed on regular servers and rely on storage (e.g., hard disk drives, solid state drives, non-volatile memory express, etc.) and network devices to service applications. These resource are by definition limited by server capacities, and thus the behavior of servers that receive more requests that they can handle should be managed. In this case servers typically serve clients proportionally to their demand and capability.

When that issue arises, storage systems should instead allocate resources fairly between requests. For example, requests coming from high performance clients can crush requests coming from less performant clients. Small requests, which are more likely to be part of some sort of control plane operation (e.g., permission verification, configuration, etc.), can be delayed in the presence of simultaneous large requests. Such small requests are typically more latency sensitive than large requests, which are more throughput sensitive. However, giving priority to small requests should not lead to an underutilization of the resource.

Figure 1 below illustrates a comparison of the behavior of traditional event-loop or thread based storage server implementations with that of the techniques described herein.

Published by Technical Disclosure Commons, 2018

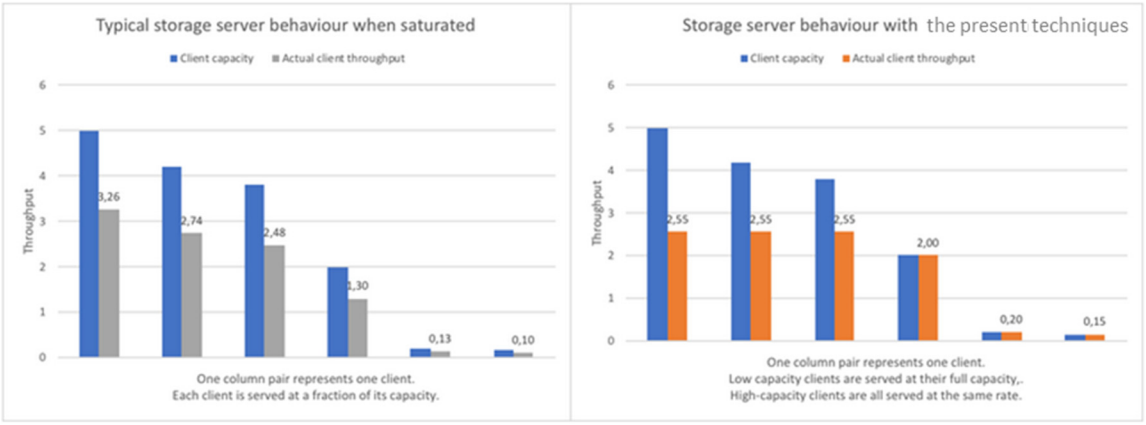

Figure 1

In both cases, the total server throughput is ten (arbitrary unit). Whereas traditional storage servers tend to slow all clients down when they are saturated, with the techniques described herein only the most demanding clients are throttled.

Described herein is a way to allocate resources optimally that ensures that small requests are treated rapidly without starving large requests.

Incoming requests may be allocated into classes. A two-pass allocation may be made to allocate resources equally between classes. The potential remaining resources may then be reallocated to classes that were not fully satisfied.

The incoming requests may be classified into classes. Typically this classification may be made between different users if the storage system has an authentication system. It may also be a segregation between e.g. data objects stored, types of data objects, storage pools, etc. Furthermore, a custom defined cost function may associate a cost to every request. The cost function may take into account the number of bytes to be read or written by a request.

The storage system may process requests in batches in the following manner. The storage system may have a parameter called a batch budget. For every new batch, the storage system may classify requests that arrived during the previous batch as well as requests that have not been completed during the previous batch. Each class has an associated request list keeping track of the costs of requests. After this classification is done, the storage system may distribute the batch budget to the classes in the following manner.

In the first phase, the batch budget is equally divided between classes. Some classes may have an allocated budget greater than the sum of the cost of their requests. The

difference is called the unused budget of this class. In the second phase, while some classes have some unused budget, this unused budget is equally redistributed between classes that do not have an unused budget (i.e., classes that fully use their budget).

Once this allocation is done, the storage system processes the requests in each class up to their allocated budget. If the sum of all requests' costs is less than the batch budget, all requests are fully processed during this batch. In this case, the next batch begins immediately to avoid resource underutilization.

Figures 2 and 3 below illustrates an example of two consecutive batches. As shown in Figure 2, at the beginning of the first batch, five requests arrive and correspond to three different classes. The three classes have their budget allocations calculated.

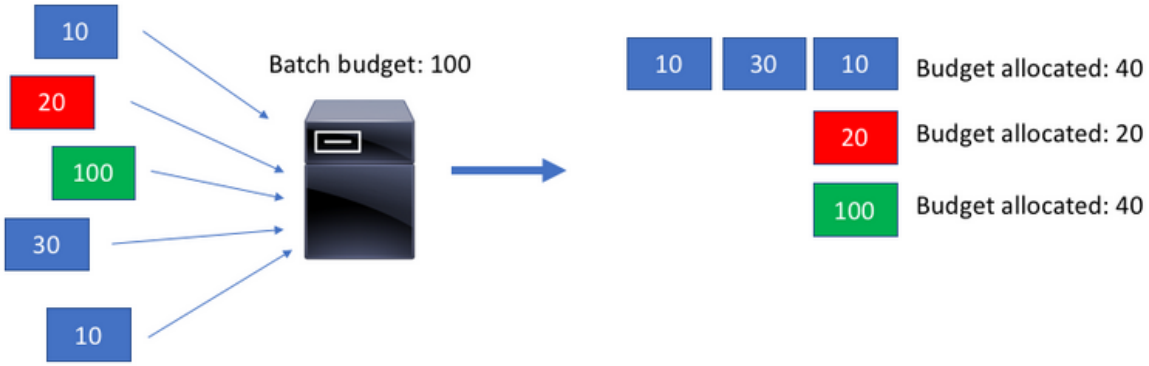

Figure 2

As shown in Figure 3, at the end of the first batch, four new requests have arrived. Two of the three classes have unfinished requests. The three classes have their budget allocation for the new batch calculated.

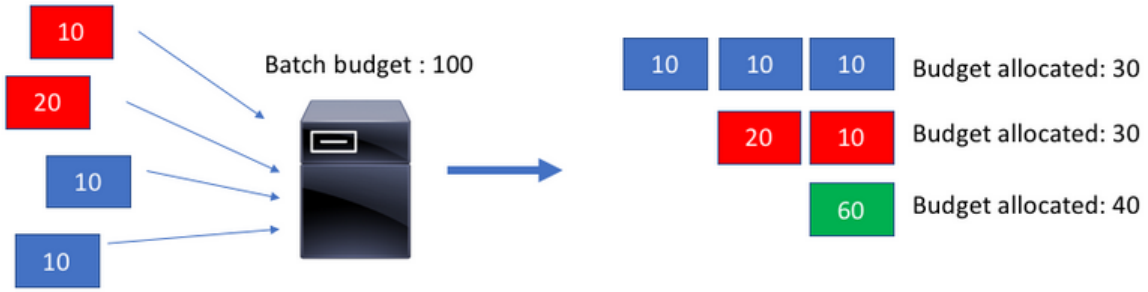

Figure 3

At the end of the second batch, the class corresponding to a single large request is not yet completed while the classes corresponding to small requests have had all their requests answered. However, increasing the batch budget has a tradeoff. It makes the

batches longer, and favors the larger requests. It also typically optimizes disk access, because disks are typically more efficient when they process few consecutive long Input/Outputs (I/Os) rather than many small random I/Os. Because the batches last longer, small requests arriving at the server while a batch is being processed have to wait for the next batch, thus increasing their effective latency. This means that increasing the batch budget optimizes the overall resource usage but penalizes small requests.

To be able to maintain a high budget to optimize the I/O ordering, a mechanism is described to allow requests arriving during a batch to "join" the current batch. A new parameter called the piggyback budget corresponds to an extra budget that is dedicated to incoming requests. If a request arrives during the processing of a batch and its cost is less than a configurable fraction of the piggyback budget (called piggyback threshold) and the remaining part of the piggyback budget, it is processed during the current batch. Otherwise, this request is processed in the next batch.

Figures 4-6 below illustrate a slightly different case than before. As shown in Figure 4, without a piggyback budget, if the batch budget is high, requests that are incoming during the first batch have to wait for the batch to finish.

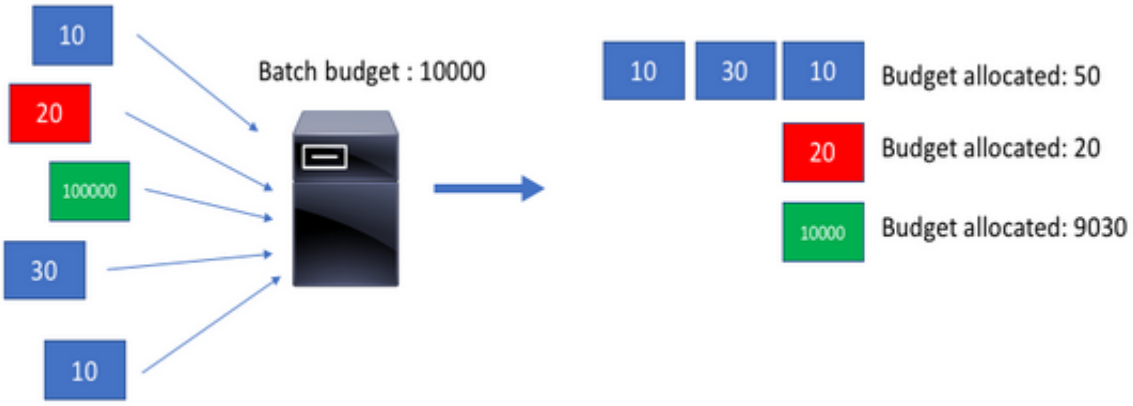

Figure 4

Figures 5 and 6 below illustrates how, with the piggyback budget, they are included in the batch.

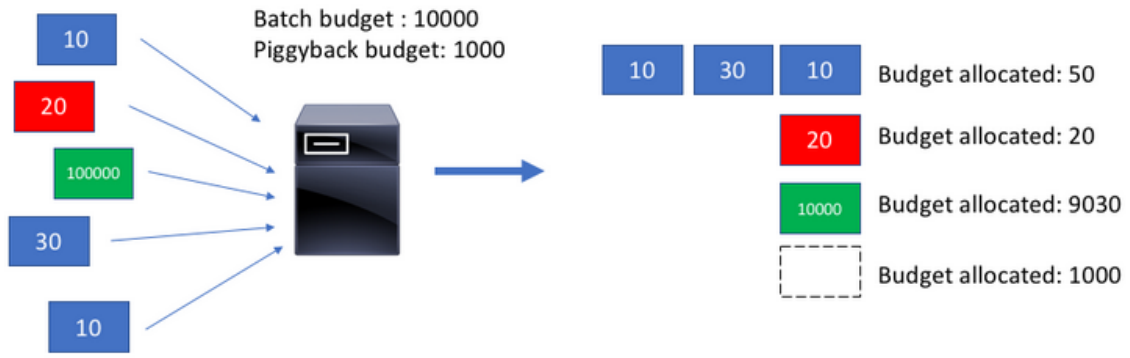

Figure 5

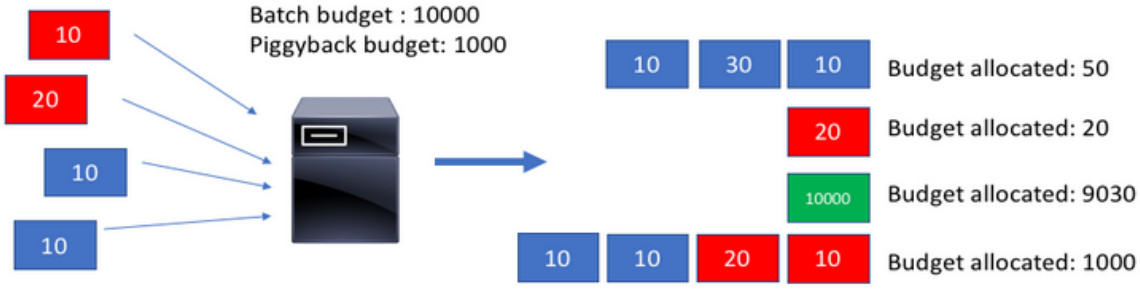

Figure 6

In the case where this piggyback budget is not fully utilized, the resource is not underused. The batch just finishes earlier and the next batch starts.

A malicious class might attempt to abuse this piggyback budget by causing it to consume more than its fair share of resources. Therefore, when an incoming request's cost is less than the piggyback threshold, the request is only added to the current batch if the sum of its cost and of the budget allocated to its class during that batch is less than the batch budget divided by the number of classes in the batch. Consequently, a class cannot gain more budget from the piggyback budget than what would have been allocated to it from the batch budget. Thus, a class does not benefit from splitting its requests versus sending it in one block.

In summary, techniques are described herein for storage systems to guarantee low latency for small requests while maintaining the system's optimal overall throughput. It batches requests, classifies them, fairly allocates resources to them, and provides a mechanism to expedite the processing of small requests. In today's cloud environments, it is critical that diverse applications run by multiple users can share access to generic storage systems without affecting each other's performance.