2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services
Mace, J. et al.
Proc. of ACM SIGCOMM '16

Reviewed by
Chun-Yu Chen and Ming zhi Yu

Summary

In this paper, the authors propose a fair queuing scheduling algorithm called 2DFQ which can be used to schedule requests from different tenants in multi-tenant cloud services and provide better resource isolation than traditional fair queuing scheduling algorithms. Traditional fair schedulers such as WFQ and WF^2Q, which have been used to schedule network packets, are unsuitable for scheduling requests in multi-tenant cloud services in that they are unable to harness the power of resource concurrency due to the fact that they are originally designed for sequential execution of requests. In addition, the fact that requests in cloud services usually have large cost variance as well as unpredictable resource cost poses challenges to traditional fair schedulers because network packets does not exhibit those two characteristics. The authors show in the paper that the result of applying these traditional fair schedulers in cloud services is a bursty schedule which results in longer and unstable request latency. To solve the problem, the authors take advantage of the concurrency of the system and design a scheduling algorithm that involves a new dimension, which is characterized by thread indices. In two sets of experiments conducted using both artificially-constructed traces and production traces from Azure Storage, 2DFQ is shown to achieve the goal of providing better resource isolation.

Highlights

The first thing we like about the paper is how the authors take advantage of resource concurrency and separate requests with different costs across different workers threads to make schedule smoother. As is mentioned in the paper, the problem of traditional fair schedulers is that they only quantify and evaluate fairness in terms of worst-case bounds. While it is true that a schedule

produced by these traditional schedulers satisfies this worst-case bound condition, the schedule tends to be bursty. Such a bursty schedule poses a challenge in cloud service requests scheduling because of the large variation existing in request cost. Under a bursty schedule, it is likely that many or all threads will be blocked for an extended period of time during which small requests will have no access to resources. To avoid producing a bursty schedule, the authors identify the cause of a bursty schedule in WF^2Q as the eligibility criteria that applies uniformly across all threads. In other words, if a request is ineligible for one thread, it is ineligible in all other threads in WF^2Q. 2DFQ is designed to break this uniformly applied criterion by making light requests eligible to run on high-index threads sooner than low-index threads so as to isolate requests of different cost. The first set of experiment conducted by the authors demonstrates the effectiveness of the adjustment made on the eligibility criterion. For example, Figure 8 (a) shows that service lag of a small tenant schedule under 2DFQ is stably close to zero which is not the case when the tenant's requests are scheduled using WFQ and WF^2Q. By taking advantages of concurrency, 2DFQ is able to produce a smooth schedule in an elegant way.

We also appreciate the way how 2DFQ deals with the issue of unknown and unpredictable resource costs. The authors point out that unlike packet scheduling where the size of a packet is known a priori, cost of requests from a tenant is usually varying and unpredictable. This poses a challenge to scheduling because in order to calculate the virtual finish time, the cost of a request is needed. And since cost of a request depends on numerous factors, using a model to estimate future cost might be inaccurate and could result in bursty schedules. 2DFQ uses a novel strategy to estimate the cost of a request and this strategy is innovative in the following aspects.

Firstly, 2DFQ performs a pessimistic cost estimation which mitigates the impact of unpredictable tenants that causes bursty schedules. The reason for performing a pessimistic cost estimation is that if multiple expensive requests are mistakenly estimated to be cheap, they will block part of the thread pool which would result in a burst of service when these expensive requests finish execution. On the contrary, if a cheap request is mistakenly estimated to be expensive, the only negative impact is a longer than-normal latency experienced by that request, and this is not

persistent due to the retroactive charging mechanism that we will cover next. In this sense, a pessimistic cost estimation is better than a optimistic cost estimation.
Secondly, the retroactive charging mechanism ensures that estimation errors will not accumulate. All estimations have errors. 2DFQ's estimation strategy is able to reconcile these errors (over a long time) through the retroactive charging mechanism. This mechanism is analogous to negative feedback used in control theory. When a request completes, the system computes the difference between the actual resource usage and the estimation and report this difference to the scheduler. The scheduler then updates the tenant's start and finish tags by this difference. For example, if a request consumes more resource than what has been estimated, the following request from the same tenant would probably be delayed to start or be moved to a thread which has smaller index. In the long run, the error between actual cost and estimated cost will be reconciled.

Lastly, we like the idea of using a technique called refresh charging to prevent a tenant from 'cheating' the scheduler. While designing the 2DFQ, the authors realized that after scheduling a series of cheap requests from a tenant, the scheduler is susceptible to underestimation if the tenant transitions to expensive requests. In other words, a tenant could cheat the scheduler every time it transitions from sending a series of cheap requests to sending an expensive request. To enable the scheduler to notice expensive request quickly, refresh charging performs an on-the-fly resource usage monitoring and charges the tenant for excess cost while a request is still running. With the three points mentioned above, 2DFQ is able to generate an accurate estimation of request cost even under the condition that request cost is varying and unpredictable.

Improvements/Extensions

We think that several aspects of the paper need improvement. The first aspect has to deal with the way how the authors conducted the second set of experiment, which is used to evaluate the performance of 2DFQ when requests are unpredictable. In section 3.2, the authors mention that one of the challenges of providing fairness in shared process is that request costs are unknown and difficult to predict. Trace of tenant T_{10} is used to illustrate the unpredictability nature of

request cost. Since 2DFQ is designed to cope with the unpredictability in request cost, it makes sense for the authors to evaluate the performance of 2DFQ under different predictability.

Nevertheless, we consider the method they used to vary the amount of workload that is predictable as questionable for the following reasons. According to the authors, the way they used to increase the amount of unpredictable workload is to make some tenants explicitly unpredictable by sampling each request pseudo-randomly from across all production traces disregarding the originating server or account. Arguably, the costs of requests generated by this method vary substantially, but how practical is it to use this kind of artificially-generated traces? When illustrating the nature of unpredictability in section 3.2, the authors used an unmodified trace of an Azure Storage tenant (Figure 4 (c)). It seems to us that there are already existing traces which they could use to serve as unpredictable workload needed in the experiment. The authors provide no explanation on why they use those randomly-sampled traces instead of existing ones. In short, we consider the way they create unpredictable workload impractical and not reflecting workload of real tenants.

We have also discovered that even using pessimistic request size prediction can fail to provide smooth scheduling. Considering a request with constantly growing size, 2DFQ will always underestimate the request size and lead to unexpected blocking. To resolve this problem, we need a more sophisticated prediction mechanism. We can improve the prediction mechanism by recording the change of request size. If the request size is decreasing, we can still use the same pessimistic approach. If the request size is constantly growing, the prediction should be more pessimistic and make the predicted size even larger than the maximum recorded size.

In the experiment for known request-cost production workloads, the results (Fig. 10) show that 2DFQ and WF^2Q have little difference for high service lag variance (the curves overlap after 75 percentile). In addition, authors have stated that 2DFQ and WF^2Q are comparable in the production workload experiments. According to these two observations, we are not convinced

that 2DFQ will be able to outperform WF^2Q in high request size variance situation (even with predictable request size) based on the results shown in the paper.

One obvious way to extend this work is to consider QoS constraint for the requests. In this paper, the authors did not consider the condition that different requests may have different QoS constraints. Please be noted that adding QoS for each request does not contradict with the "fairness" in fair queueing since fair queuing is focusing on the fairness between tenants, not the requests. Because 2DFQ tends to schedule requests with larger size to lower index threads, this naturally leads to longer blocking time for those requests with larger size. Once there are tight timing constraints applied to those requests, they will probably have more probability to fail to meet those constraints.

Another dimension of timing constraints may be helpful to enhance the existing 2DFQ algorithm (it should be called 3DFQ in this case). One way to do this is to divide threads into groups with same number of threads in each group. For group selection, the mechanism will be different from the original 2DFQ that tends to push requests with similar size to the same threads. Instead, it should assign requests with tight timing constraints into different groups when considering timing constraints. By doing so, we can make sure that requests with tight timing constraints will not compete to have the same source.

We can then use the 2DFQ mechanism with timing constraints to decide which thread in the group the request should be assigned to. In other words, in addition to subtracting the function of thread index when calculating virtual start time, a function of timing constraints may be added to further separate requests into different threads. For example, we can further subtract a factor that is weighted inversely to the timing constraint from the finish time.

The goal of this paper is to produce smooth schedule without long blocking time by reducing burstly condition. One way to further improve this property is to separate predictable and unpredictable requests into different threads. By doing so, we can ensure that the predictable requests will not be affected by unpredictable requests. This will prevent requests with constant growing size to have influence on those requests with predictable size.

As mentioned earlier, if we have the statistics of what is the size changing tendency of certain requests, we can further assign those requests with growing size to the threads for unpredictable requests and those with decreasing size to the threads for predictable requests with pessimistic size estimation.

The last improvement we consider is to extend the work to support preemptive scheduling. Other than the original preemption based on certain priority criteria, we can punish the unpredictable requests with low priorities by canceling their execution after they have exceeded certain amount of expected execution time to ensure the smoothness of the scheduling.