2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services

Jonathan Mace[1], Peter Bodik[2], Madanlal Musuvathi[2], Rodrigo Fonseca[1], Krishnan Varadarajan[2]

[1]Brown University, [2]Microsoft

ABSTRACT

In many important cloud services, different tenants execute their requests in the thread pool of the same process, requiring fair sharing of resources. However, using fair queue schedulers to provide fairness in this context is difficult because of high execution concurrency, and because request costs are unknown and have high variance. Using fair schedulers like WFQ and WF^{2}Q in such settings leads to *bursty schedules*, where large requests block small ones for long periods of time. In this paper, we propose Two-Dimensional Fair Queuing (2DFQ), which spreads requests of different costs across different threads and minimizes the impact of tenants with unpredictable requests. In evaluation on production workloads from Azure Storage, a large-scale cloud system at Microsoft, we show that 2DFQ reduces the burstiness of service by 1-2 orders of magnitude. On workloads where many large requests compete with small ones, 2DFQ improves 99th percentile latencies by up to 2 orders of magnitude.

CCS Concepts

•**Networks** → **Cloud computing;** *Packet scheduling;*
•**Computer systems organization** → *Availability;*

Keywords

Fair Request Scheduling; Multi-Tenant Systems

1. INTRODUCTION

Many important distributed systems and cloud services execute requests of multiple tenants simultaneously. These include storage, configuration management, database, queuing, and co-ordination services, such as Azure Storage [9], Amazon Dynamo [16], HDFS [53], ZooKeeper [36], and many more. In this context, it is crucial to provide resource isolation to ensure that a single tenant cannot get more than its fair share of resources, to prevent aggressive tenants or unpredictable workloads from causing starvation, high latencies, or reduced throughput for others. Systems in the past have suffered cascading failures [19,27], slowdown [14,20,27,28,33], and even cluster-wide outages [14,19,27] due to aggressive tenants and insufficient resource isolation.

SIGCOMM '16, August 22 - 26, 2016, Florianopolis , Brazil

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934878

However, it is difficult to provide isolation in these systems because multiple tenants execute *within the same process*. Consider the HDFS NameNode process, which maintains metadata related to locations of blocks in HDFS. Users invoke various APIs on the NameNode to create, rename, or delete files, create or list directories, or look up file block locations. As in most shared systems, requests to the NameNode wait in an admission queue and are processed in FIFO order by a set of worker threads. In this setting tenant requests contend for resources, such as CPU, disks, or even locks, from *within* the shared process. As a result, traditional resource management mechanisms in the operating system and hypervisor are unsuitable for providing resource isolation because of a mismatch in the management granularity.

In many domains, resource isolation is implemented using a fair queue scheduler, which provides alternating service to competing tenants and achieves a fair allocation of resources over time. Fair schedulers such as Weighted Fair Queuing [46], which were originally studied in the context of packet scheduling, can be applied to shared processes since the setting is similar: multiple tenants submit flows of short-lived requests that are queued and eventually processed by a server of limited capacity. However, in shared processes there are three additional challenges that must be addressed:

- **Resource concurrency:** Thread pools in shared processes execute many requests concurrently, often tens or even hundreds, whereas packet schedulers are only designed for sequential execution of requests (*i.e.* on a network link);
- **Large cost variance:** Request costs vary by at least 4 orders of magnitude across different tenants and API types, from sub-millisecond to many seconds. By contrast, network packets only vary in length by up to 1.5 orders of magnitude (between 40 and 1500 bytes). Unlike CPU thread schedulers, requests are not preemptible by the application;
- **Unknown and unpredictable resource costs:** The execution time and resource requirements of a request are not known at schedule time, are difficult to estimate up front, and vary substantially based on API type, arguments, and transient system state (*e.g.*, caches). By contrast, the length of each network packet is known a priori and many packet schedulers rely on this information.

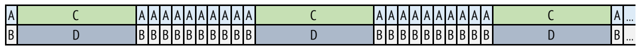
These challenges affect the quality of schedules produced by algorithms such as Weighted Fair Queuing (WFQ) [46] and Worst-Case Fair Weighted Fair Queuing (WF^{2}Q) [6]. Figure 1a shows an example of a bursty request schedule which does occur in practice as we verify in our evaluation. In the bursty schedule, the service rates allocated to tenants A and B oscillate significantly because C and D have large requests. Most fair packet schedulers produce the bursty schedule, despite the existence of the better smooth schedule. This occurs in part because the schedulers only quantify and evaluate fairness in terms of *worst-case* bounds, which the bursty and smooth schedules both satisfy. For example, consider MSF^{2}Q [8], a packet scheduler that extends WF^{2}Q to multiple aggregated network links (a setting analogous to request scheduling in worker thread pools). MSF^{2}Q bounds by how much a tenant can fall behind its fair share to $N \cdot L_{max}$ where N is the number of threads and L_{max} is the cost of the largest request. It also bounds by how much a tenant i can get ahead of its fair share to $N \cdot L^i_{max}$ where L^i_{max} is the cost of i's largest request. Worst-case bounds are sufficient to avoid unacceptable bursts in packet scheduling, but in our context they are insufficient due to large concurrency (large N) and large cost variance (large L_{max}). It might not be possible to improve worst case bounds in theory, so instead we seek a scheduler that, in practice, achieves smoother schedules on average.

The practical obstacle to smooth schedules is cost estimation. In shared services, request costs are not known at schedule time; instead the scheduler must estimate costs based on past requests or some other model. However, request costs are difficult to predict and estimates could be off by orders of magnitude. When a tenant sends many expensive requests estimated to be very cheap, the scheduler can start them together, blocking many or all available threads for long periods of time. Thus incorrect costs lead to bursty schedules and high latencies, particularly for tenants with small requests.

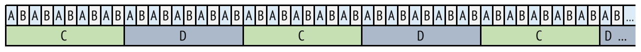
In this paper, we present Two-Dimensional Fair Queuing (2DFQ)[1], a request scheduling algorithm that produces fair and smooth schedules in systems that can process multiple requests concurrently. Our solution builds on two insights to address the challenges above. First, we take advantage of the concurrency of the system and *separate requests with different costs across different worker threads*. This way, large requests do not take over all the threads in the system and do not block small requests for long periods of time. Second, when request costs are unknown a priori, we use *pessimistic cost estimation* to co-locate unpredictable requests with expensive requests, keeping them away from tenants with small and predictable requests for whom they would cause bursty schedules.

2DFQ produces smooth schedules like the schedule illustrated in Figure 1b, even in the presence of expensive or unpredictable tenants. 2DFQ improves per-tenant service rates compared to existing schedulers such as WFQ, WF^{2}Q and MSF^{2}Q. While it keeps the same worst-case bounds as MSF^{2}Q, 2DFQ produces better schedules in the average case by avoiding bursty schedules where possible.

[1]Two-dimensional because it schedules requests across both time and the available threads.



(a) Bursty schedule


ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB...
C
D
C
D
C
D ...

(b) Smooth schedule

Figure 1: An illustration of request execution over time with four tenants sharing two threads. Tenants A and B send requests with 1 second duration while tenants C and D with 10 second duration. In each schedule, rows represent the threads over time labeled by the currently executing tenant. Both schedules are fair; over long time periods, all tenants receive their fair share. Top: *bursty* schedule; small requests receive no service for 10 second periods. Bottom: *smooth* schedule with only 1 second gap between two requests of tenant A and B. Schedulers such as WFQ, WF^{2}Q, or MSF^{2}Q generate the bursty schedule; 2DFQ is designed to generate the smooth schedule.

The contributions of this paper are as follows:

- Using production traces from Azure Storage [4, 9], a large-scale system deployed across many Microsoft datacenters, we demonstrate scheduling challenges arising from high concurrency and variable, unpredictable request costs;
- We improve upon existing fair schedulers with Two-Dimensional Fair Queuing (2DFQ), a request scheduler based on WF^{2}Q that avoids bursty schedules by biasing requests of different sizes to different threads;
- To handle unknown request costs we present 2DFQ^E, which extends 2DFQ's cost-based partitioning with pessimistic cost estimation to mitigate the impact of unpredictable tenants that cause bursty schedules;
- We evaluate 2DFQ and 2DFQ^E with extensive simulations based on production workload traces from Azure Storage. We find that 2DFQ has up to 2 orders of magnitude less variation in service rates for small and medium requests compared to WFQ and WF^{2}Q. Across a suite of 150 experiments, 2DFQ^E dramatically reduces mean and tail latency, by up to 2 orders of magnitude for predictable workloads when they contend against large or unpredictable requests.

2. MOTIVATION

Need for Fine-Grained Resource Isolation. Many important datacenter services such as storage, database, queuing, and coordination services [9, 30, 36, 53], are shared among multiple tenants simultaneously, due to the clear advantages in terms of cost, efficiency, and scalability. In most of these systems, multiple tenants contend with each other for resources *within* the same shared processes. Examples include performing metadata operations on the HDFS NameNode and performing data operations on HDFS DataNodes [53].

When tenants compete inside a process, traditional and well-studied resource management techniques in the operating system and hypervisor are unsuitable for protecting tenants from each other. In such cases, aggressive tenants can overload the process and gain an unfair share of resources. In the extreme, this lack of isolation can lead to a denial-of-service to well-behaved tenants and even system wide outages. For example, eBay Hadoop clusters regularly suffered denial of service attacks caused by heavy users overloading the shared HDFS NameNode [20, 34]. HDFS users report slowdown for a variety of reasons: poorly written jobs making many API calls [33]; unmanaged, aggressive background tasks making too many concurrent requests [32]; and computationally expensive APIs [28]. Impala [37] queries can fail on overloaded Kudu [41] clusters due to request timeouts and a lack of fair sharing [38]. Cloudstack users can hammer the shared management server, causing performance issues for other users or even crashes [14]. Guo et al. [27] describe examples where a lack of resource management causes failures that cascade into system-wide outages: a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading servers; overloaded servers not responding to heartbeats, triggering further data replication and overload.

Given the burden on application programmers, inevitably, many systems do not provide isolation between tenants, or only utilize ad-hoc isolation mechanisms to address individual problems reported by users. For example, HDFS recently introduced priority queuing [29] to address the problem that *"any poorly written MapReduce job is a potential distributed denial-of-service attack,"* but this only provides coarse-grained throttling of aggressive users over long periods of time. CloudStack addressed denial-of-service attacks in release 4.1, adding manually configurable upper bounds for tenant request rates [13]. A recent HBase update [31] introduced rate limiting for operators to throttle aggressive users, but it relies on hard-coded thresholds, manual partitioning of request types, and lacks cost-based scheduling. In these examples, the developers identify multi-tenant fairness and isolation as an important, but difficult, and as-yet unsolved problem [10, 38, 47].

Research projects such as Retro [43], Pulsar [3], Pisces [52], Cake [60], IOFlow [54], and more [61], provide isolation in distributed systems using rate limiting or fair queuing. Rate limiters, typically implemented as token buckets, are not designed to provide fairness at short time intervals. Depending on the token bucket rate and burst parameters, they can either underutilize the system or concurrent bursts can overload it without providing any further fairness guarantees. Fair queuing is an appealing approach to provide fairness and isolation because it is robust to dynamic workloads. However, as we demonstrate in §3, in many systems, request costs can vary by up to four orders of magnitude and are unpredictable, which can cause head-of-line blocking for small requests and significantly increase latencies.

Desirable Properties. In this paper, we characterize a resource isolation mechanism that provides "soft" guarantees by using a fair scheduler. The scheduler attempts to share the resources available within a process equally or in proportion to some weights among tenants currently sending requests to the system. Incoming requests are sent to (logical) per-tenant queues. The system runs a set of worker threads, typically in the low 10s, but sometimes in the 100s, to process these requests. When a worker thread is idle, it picks the next request from one of the per-tenant queues based on a scheduling policy that seeks to provide a fair share to each of the tenants.

We specify two desirable properties of such a scheduler. First and foremost, the scheduler should be *work conserving* — a worker thread should always process some request when it becomes idle. This property ensures that the scheduler maximizes the utilization of resources within a datacenter. This requirement precludes the use of ad-hoc throttling mechanisms to control misbehaving tenants.

Second, the scheduler should not be *bursty* when servicing different tenants. For example, a scheduler that alternates between extended periods of servicing requests from one tenant and then the other is unacceptable even though the two tenants get their fair share in the long run. Providing fairness at smaller time intervals ensures that tenant request latency is more stable and predictable, and mitigates the aforementioned challenges like denial-of-service and starvation.

Fair Queuing Background. A wide variety of packet schedulers have been proposed for fairly allocating link bandwidth among competing network flows. Their goal is to approximate the share that would be provided under Generalized Processor Sharing (GPS) [46], constrained in that only one packet can be sent on the network link at a time, and that packets must be transmitted in their entirety. Well-known algorithms include Weighted Fair Queueing (WFQ) [46], Worst-case Fair Weighted Fair Queueing (WF^2Q) [6], Start-Time Fair Queueing (SFQ) [23], Deficit Round-Robin (DRR) [50], and more. We briefly describe WFQ.

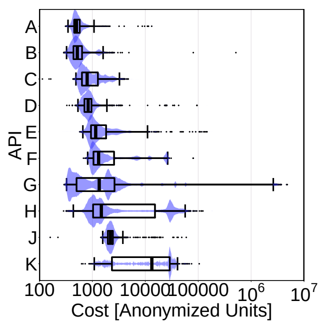
WFQ keeps track of the work done by each tenant over time and makes scheduling decisions by considering the work done so far and cost of each tenant's next request. To track fair share, the system maintains a *virtual time* which increases by the rate at which backlogged tenants receive service, for example, for 4 tenants sharing a worker thread of capacity 100 units per second, virtual time advances at a rate of 25 units per second; for 4 tenants sharing two worker threads each with capacity 100 units per second, virtual time advances at a rate of 50 units per second; and so on. We use $A(r_f^j)$ to denote the wallclock arrival time of the j^{th} request of tenant f at the server, and $v(A(r_f^j))$ to denote the system virtual time when the request arrived. WFQ stamps each request with a virtual start time $S(r_f^j)$ and virtual finish time $F(r_f^j)$ as follows:

$$S(r_f^j) = max\{v(A(r_f^j)), F(r_f^{j-1})\} \qquad F(r_f^j) = S(r_f^j) + \frac{l_f^j}{\phi_f}$$

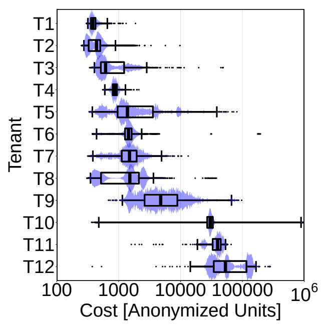
where ϕ_f is the weight of tenant f and l_f^j is the size of the request. For a single tenant, the start time of the j^{th} request is simply the finish time of the $(j-1)^{th}$ request, unless the tenant was inactive, in which case it fast-forwards to the current system virtual time. Each time a thread is free to process a request, WFQ schedules the pending request with the lowest virtual finish time.

Worst-case Fair Weighted Fair Queuing (WF^2Q) [6] extends WFQ and is widely considered to have better fairness bounds. The authors identify and address a prominent cause of bursty schedules that can occur on a single networking link. WF^2Q restricts WFQ to only schedule requests after they become *eligible*, with a request becoming eligible only if it would have begun service in the corresponding GPS system, i.e. $S(r) \leq v(now)$.

Request scheduling across many threads is analogous to packet scheduling across multiple aggregated links. Blanquer and Özden previously extended WFQ to multiple aggregated links and examined the changes to its fairness bounds, packet delays, and work conservation [8]. While they termed the algorithm MSFQ, we retain the name WFQ in the interest of familiarity, and use WF^2Q to refer to the naïve work conserving extension of WF^2Q to multiple aggregated links.

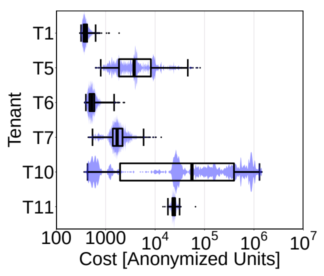


(a) Cost distributions for 10 APIs.



(b) Cost distributions for 12 tenants.

Figure 2: Measurements of Azure Storage show widely varying request costs. Whiskers extend to 1st and 99th percentiles; violins show distribution shape.



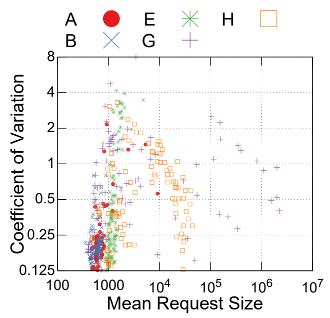


Figure 3: Left: cost distributions of some tenants using API G, illustrating variability across tenants. Right: scatter plot showing, for each tenant and API, the average request cost (x-axis) and coefficient of variation (y-axis) for the tenant's use of that API. Each point represents one tenant on one API, indicated by color. Each API has tenants using it in predictable and unpredictable ways.

3. CHALLENGES

In this section we describe two challenges to providing fairness in shared processes. The first challenge, described in §3.1, arises when requests with large cost variance are scheduled across multiple threads concurrently. The second challenge, described in §3.2, arises when request costs are unknown and difficult to predict. To demonstrate the challenges we collect statistics from 5-minute workload samples across 50 production machines of Azure Storage [4, 9], a large-scale system deployed across many Microsoft datacenters.

3.1 High Request Cost Variability

We first illustrate how request costs in shared services vary widely, by up to 4 orders of magnitude. For cost, we report the CPU cycles spent to execute the request, and anonymize the units. Other metrics we considered include wallclock execution time and dominant resource cost [21].

Figure 2a shows anonymized cost distributions for several different APIs in Azure Storage, illustrating how some APIs are consistently cheap (A), some vary widely (K), and some are usually cheap but occasionally very expensive (G).

Figure 2b shows cost distributions for several different tenants of Azure Storage, illustrating how some tenants only make small requests with little variation (T_1), some tenants make large requests but also with little variation (T_{11}), and some tenants make a mixture of small and large requests with a lot of variation (T_9).

In aggregate across all tenants and APIs, request costs span four orders of magnitude — a much wider range than network packets, for which most scheduling algorithms were developed, where sizes only vary by 1.5 orders of magnitude (between 40 and 1500 bytes). High cost variance is not unique to this production system and is shared by many popular open-source systems as well: in storage and key-value stores, users can read and write small and large objects; in databases, users can specify operations that scan large tables; in configuration and metadata services, users can enumerate large lists and directories. All of these operations can have very high cost compared to the average operations in the system.

As illustrated in Figure 1, both bursty and smooth schedules are possible when there are multiple worker threads. Bursty schedules adversely affect tenants with small requests by servicing them in high-throughput bursts rather than evenly paced over time. Since realistic systems have such high cost variance, tenants like T_1 will experience large service oscillations if the scheduler lets too many expensive requests occupy the thread pool. In §6 we verify that existing schedulers like WFQ and WF^{2}Q do produce bursty schedules in practice.

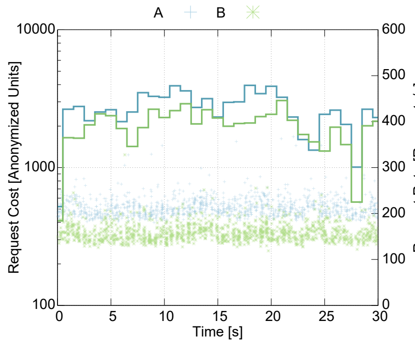
Our insight to generating smooth schedules is that given the large number of available threads, we can spread requests of different costs across different threads. In some cases it will be preferable to prioritize small requests in order to prevent long periods of blocking that would occur if we selected a large request. We discuss the details of our approach in §4.

3.2 Unknown Request Costs

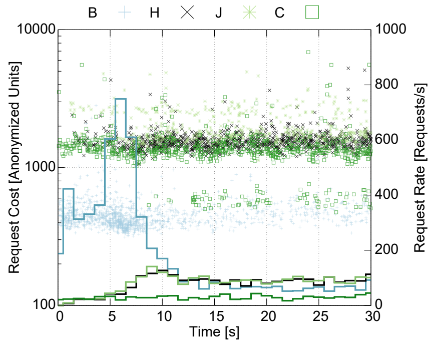
To motivate the second challenge we illustrate how some tenants are dynamic, with varying and unpredictable request costs. Figure 4 shows time series for T_2, T_3, and T_{10}, illustrating request rates and costs for the APIs being used. T_2 (4a) has a stable request rate, small requests, and little variation in request cost. T_3 (4b) submits a large burst of requests that then tapers off, with costs across four APIs that vary by about 1.5 orders of magnitude. T_{10} (4c) is the most unpredictable tenant, with bursts and lulls of requests, and costs that span more than three orders of magnitude.

Even within each API, request costs vary by tenant. For example, while API G illustrated in Figure 2a has several orders of magnitude between its 1% and 99% request costs, if we also condition on the tenant, see Figure 3 (left), most tenants using this API actually have very low cost variance. Figure 3 (right) shows the scatter plot of mean and coefficient of variation (CoV = mean / stdev) of request costs across many tenants and APIs. The figure illustrates that each API has tenants using it in predictable and unpredictable ways.

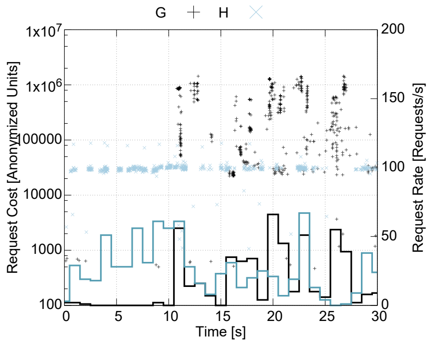
Unknown request costs are a challenge to making effective scheduling decisions, since packet schedulers need to know costs a priori. As a workaround, costs can be estimated based on past requests or some other model. However, while cost estimation is suitable for stable, predictable tenants, it loses effectiveness for dynamic tenants. Models can be inaccurate since costs depend on numerous factors: the API being called, its parameters, content of various caches, sizes of internal objects that the request might process, etc. Estimates based on recent request history, such as moving averages, only reflect the past and can be consistently wrong proportional to how frequently costs vary.



(a) Stable tenant (T_2)



(b) Stable with gradual changes (T_3)



(c) Unstable tenant with frequent changes (T_{10})

Figure 4: Details of three Azure Storage tenants over a 30 second interval. Each color represents a different API labeled consistently with Figure 2a. Points represent individual requests with their costs on the left y-axis. Lines represent the aggregate request rate during 1 second time intervals (right y-axis).

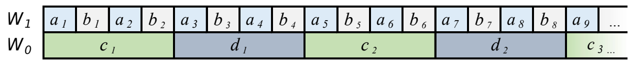
Incorrect cost estimates lead to bursty schedules. They occur when an expensive request *masquerades* as a cheap request and is scheduled as though it is cheap, blocking its worker thread for longer than expected. When a burst of these requests occurs, they can block many or all worker threads for a long period of time, impacting the latency of all other waiting tenants. The challenge is orthogonal to the scheduling strategy – it affects not only existing schedulers, but also new schedulers designed to address the cost variability challenge of §3.1.

Feedback delays exacerbate the impact of underestimates. Consider an estimator based on per-tenant moving averages, a typical approach to estimating request costs [3, 42, 43, 51, 52]. There is an inherent feedback delay between estimating a request's cost and eventually updating the estimator with the actual cost once the request finishes. If a tenant transitions from cheap to expensive requests then the scheduler will incorrectly schedule not just one request, but potentially up to N requests (where N is the number of threads), since the expensive costs won't be reflected back in the estimator until *after* one of the expensive requests has completed. While a bursty schedule is inevitable when a tenant transitions from cheap to expensive requests, in this scenario it can be significantly amplified. The tenant will also deviate from its fair share unless the scheduler reconciles the error between each request's estimated and actual cost.

Our insight to generating smooth schedules under unknown request costs stems from the following observations. If the scheduler underestimates a request's cost, then the request can block the thread pool for a long period of time leading to bursty schedules for other tenants. However, if the scheduler overestimates a request's cost, it only immediately affects the one tenant that was forced to wait for longer than it should have. Since workloads typically contain a mixture of predictable and unpredictable tenants, it is better to give good service to a predictable tenant than to try – and fail – to give good service to an unpredictable tenant. In order to prevent unpredictable tenants from interfering with predictable tenants, we try to reduce the chance of underestimating request costs. We discuss the details of our approach in §5.

4. TWO-DIMENSIONAL FAIR QUEUING

The goal of Two-Dimensional Fair Queuing (2DFQ) is to produce smooth schedules for tenants with small requests, by minimizing burstiness over time and over space. This section outlines the design of 2DFQ, a request scheduler for known request costs; in §5 we present 2DFQ^E, a scheduler for unknown request costs.


W_1
a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 a_7 b_7 a_8 b_8 a_9 ...
W_0
c_1 d_1 c_2 d_2 c_3 ...

(a) Ideal request schedule over time on two threads

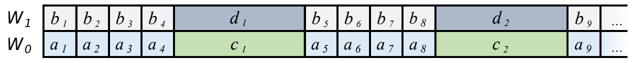
Request	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	
Start Time	0	1	2	3	4	5	6	7	8	
Finish Time	1	2	3	4	5	6	7	8	9	...

Request	c_1	c_2	c_3	
Start Time	0	4	8	
Finish Time	4	8	12	...

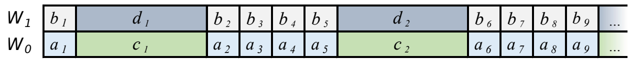
Request	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	
Start Time	0	1	2	3	4	5	6	7	8	
Finish Time	1	2	3	4	5	6	7	8	9	...

Request	d_1	d_2	d_3	
Start Time	0	4	8	
Finish Time	4	8	12	...

(b) Request start and finish times for WFQ and WF²Q


W_1
b_1 b_2 b_3 b_4 d_1 b_5 b_6 b_7 b_8 d_2 b_9 ...
W_0
a_1 a_2 a_3 a_4 c_1 a_5 a_6 a_7 a_8 c_2 a_9 ...

(c) Request schedule produced under WFQ


W_1
b_1 d_1 b_2 b_3 b_4 b_5 d_2 b_6 b_7 b_8 b_9 ...
W_0
a_1 c_1 a_2 a_3 a_4 a_5 c_2 a_6 a_7 a_8 a_9 ...

(d) Request schedule produced under WF²Q

Figure 5: Example schedules comparing WFQ and WF²Q to the ideal schedule for four tenants sharing two worker threads. Tenants A and B have request size 1; tenants C and D have request size 4. See description in §4.

We begin by demonstrating how WFQ and WF²Q produce bursty schedules. Consider four backlogged tenants ($A \ldots D$) sharing two worker threads (W_0 and W_1). Tenants A and B have small requests (size 1), while tenants C and D have large requests (size 4). Figure 5a illustrates an ideal schedule of requests over threads in this scenario.

Figure 5b outlines the virtual start and finish times used by WFQ and WF²Q. Figure 5c illustrates the resulting schedule for WFQ. Since WFQ schedules requests in ascending order of finish time, it uses both threads to execute 4 requests each for A and B. Only at $t = 4$ do C and D have the lowest finish time causing WFQ to simultaneously execute one request each for C and D, occupying the thread pool until $t = 8$. This schedule is bursty for A and B, because they each get a period of high throughput followed by a period of zero throughput. In general, WFQ's bursts are proportional to the maximum request size (*i.e.*, the size of C and D's requests) and the number of tenants present (*i.e.*, doubling the number of tenants would double the period of blocking).

Figure 5d illustrates the resulting schedule under WF²Q which also has periods of blocking proportional to maximum request size. WF²Q also schedules requests in ascending order of finish time, but with the additional condition that a request cannot be scheduled if its virtual start time has not yet arrived. As a result, WF^{2}Q does not schedule the second requests of A or B – their virtual start time is 1, which means they cannot be scheduled until $t = 2$. The only remaining requests for WF^{2}Q to pick are those of C and D. Like WFQ, WF^{2}Q produces a bursty schedule that alternates between concurrent service for A and B, followed by concurrent service for C and D.

Burstiness occurs under WF^{2}Q when multiple worker threads become available and only large requests are eligible to be scheduled. Since each request is instantaneously eligible to run on all worker threads when its virtual start time $S(r_j)$ arrives, if small requests are ineligible for one thread then they will be ineligible for all threads. The key to 2DFQ is to break this tie. 2DFQ modifies WF^{2}Q's eligibility criterion to make a request eligible at different times for different worker threads, avoiding WF^{2}Q's "all or nothing" behavior. 2DFQ uniformly staggers each request's eligibility across threads, making them eligible to run on high-index threads sooner than low-index threads. Formally, in a system with n threads, r_j is eligible on thread i at virtual time $S(r_j) - \frac{i}{n} \times l_j$ where $0 \le i < n$. This staggers the eligibility of r_j across threads in intervals of $\frac{l_j}{n}$. Once $S(r_j)$ arrives, r_j will be eligible on all worker threads.

With these modified eligibility criteria, each worker thread has a different threshold for when a request will become eligible; while one thread might find that only large requests are eligible, other threads might still see eligible small requests and select those instead. The practical effect of 2DFQ is to partition requests across threads by size. Small requests become eligible on high-index threads first and tend to be dequeued and serviced on those threads before they are ever eligible for low-index threads. On the other hand, due to the lack of eligible small requests, low-index threads end up mostly servicing large requests.

Returning to the previous example, Figure 6a outlines the modified eligibility times of requests under 2DFQ, which are now different for W_0 and W_1. Figure 6b illustrates the schedule under 2DFQ. This time, the second requests of A and B will not be eligible on W_0 until $t = 2$, but on W_1 they *are* eligible at $t = 1$. As a result, 2DFQ schedules c_1 on W_0, but selects a_2 for W_1. Thereafter, A and B continue alternating requests on W_1.

The following theorem shows that 2DFQ is fair by proving a bound on how far tenants can fall behind the fair share provided by the ideal fluid GPS server.

Theorem 1. *At any instant t, if* $W^f_{2DFQ}(0,t)$ *represents the amount of resources consumed by tenant* f *under 2DFQ and* $W^f_{GPS}(0,t)$ *represents the resources consumed by* f *under GPS, then*

$$W^f_{GPS}(0,t) - W^f_{2DFQ}(0,t) \le N \times L^{max}$$

where N is the number of threads and L^{max} *is the maximum resource consumed by any request in the system.*

Proof. The proof follows from the corresponding theorem for MSFQ (Theorem 3 in [8]) and the fact that adding a regulator does not modify this bound (Theorem 1 in [6]) provided the regulator makes a request eligible for schedule at or before the start time of the request in the GPS server. The eligibility condition of 2DFQ has this property. □

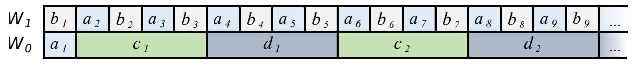
Request	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	
Eligible W_0	0	1	2	3	4	5	6	7	8	
Eligible W_1	-0.5	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	...

Request	c_1	c_2	c_3	
Eligible W_0	0	4	8	
Eligible W_1	-2	2	6	...

Request	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	
Eligible W_0	0	1	2	3	4	5	6	7	8	
Eligible W_1	-0.5	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	...

Request	d_1	d_2	d_3	
Eligible W_0	0	4	8	
Eligible W_1	-2	2	6	...

(a) Modified eligibility times under 2DFQ



W_1
b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 a_7 b_7 a_8 b_8 a_9 b_9 ...
W_0
a_1 c_1 d_1 c_2 d_2 ...

(b) Request schedule produced under 2DFQ.

Figure 6: Schedule for 2DFQ for the example described in §4 and Fig. 5.

5. SCHEDULING REQUESTS WITH UNKNOWN COST

This section outlines the design of 2DFQ^E, which extends 2DFQ with mechanisms for effectively dealing with unknown request costs. We outline the pessimistic estimation strategy of 2DFQ^E as well as two practical bookkeeping mechanisms – retroactive charging and refresh charging – for managing unpredictable request costs.

Bookkeeping: Retroactive Charging Since request costs aren't known a priori, most schedulers update counters at schedule time using an *estimate* of the cost, for example using a moving average that is updated when requests complete. For schedulers based on virtual time (cf. §2) this is l_r, used to calculate finish times: $F(r) = S(r) + l_r/\phi_f$.

Scheduling only based on estimates can lead to arbitrary unfairness in systems where multiple requests can execute concurrently. For example, suppose we use the cost of the most recently completed request as our estimate. Then in a thread pool with n threads, any tenant can achieve approximately n times their fair share by alternating between one small request of size 1, followed by n concurrent large requests of size k. The scheduler would predict size n for the small request, and size 1 for the large requests, thereby charging $n + k$ for $kn + 1$ work.

Retroactive charging ensures that a tenant is eventually charged for the resources it actually consumed and not just our a priori estimate. The system measures the resource usage c_r of each request, and reports it back to the scheduler upon completion. If $c_r > l_r$, we charge the tenant for its excess consumption; if $c_r < l_r$, we refund the tenant for unused resources. To do this we adjust the tenant's start and finish tags by $c_r - l_r$. Regardless of the initial estimate l_r, retroactive charging eventually reconciles the error between l_r and c_r, thereby guaranteeing that the tenant will receive its true fair share of resources in the long run.

Pessimistic Cost Estimation It is important to have a good cost estimator to avoid bursty schedules in the short term. Following from §3.2, we are most concerned about the case when our cost estimate for a tenant is low and it transitions to expensive requests; *i.e.* $l_r \ll c_r$. If the scheduler mistakenly estimates an expensive request to be cheap, the expensive request can occupy a worker thread for significantly longer than the scheduler may have anticipated. If the scheduler mistakenly estimates multiple expensive requests to be cheap, then they can concurrently block part of all of the thread pool. This will cause a burst of service to this tenant followed by a lull once the scheduler incorporates the true cost and compensates other tenants for this aggressive allocation. The duration

```
1: procedure Enqueue($r_f^j$)                          ▷ request $j$ of tenant $f$
2:     if $f \notin$ A then
3:         A $\leftarrow$ A $\cup f$
4:         $S_f \leftarrow \max(S_f, v(\text{now}))$
5:     end if
6:     Push($Q_f, r_f^j$)                               ▷ enqueue $r_f^j$
7: end procedure

8: procedure Refresh                                   ▷ runs periodically
9:     for $r_f^j \in$ Running requests do
10:        $c \leftarrow$ new resource usage
11:        if $c < c_f^j$ then                          ▷ already paid for $c$
12:            $c_f^j \leftarrow c_f^j - c$
13:        else                     ▷ not paid yet, charge the tenant
14:            $S_f \leftarrow S_f + (c - c_f^j)/\phi_f$
15:            $c_f^j \leftarrow 0$
16:        end if
17:    end for
18: end procedure

19: procedure Dequeue($i$)                              ▷ dequeue to thread $i$
20:     $E_{now} \leftarrow \{f \in \text{A} : S_f - \frac{i}{n} L_{max}^f < v(\text{now})\}$
21:     $f^* \leftarrow f \in E_{now}$ with smallest $S_f + L_{max}^f/\phi_f$
22:     $r_{f^*}^j \leftarrow$ Pop($Q_{f^*}$)                     ▷ request to run
23:     $c_{f^*}^j = L_{max}^{f^*}$          ▷ remember how much we paid for $r_{f^*}^j$
24:     $S_{f^*} = S_{f^*} + L_{max}^{f^*}/\phi_{f^*}$
25:     return $r_{f^*}^j$
26: end procedure

27: procedure Complete($r_f^j$)                   ▷ Service was completed
28:     $c \leftarrow$ new resource usage
29:     $T \leftarrow$ total resource usage of $r_f^j$
30:     $L_{max}^f \leftarrow \max(\alpha L_{max}^j, T)$                 ▷ update $L_{max}^j$
31:     $S_f \leftarrow S_f + (c - c_f^j)/\phi_f$                 ▷ reconcile usage
32:     if $|Q_f| = 0$ then
33:         A $\leftarrow$ A $\setminus f$
34:     end if
35: end procedure
```

Figure 7: 2DFQ^E on n threads. A is the set of active tenants. c_f^j keeps track of how much credit we have left for this request. It is initialized on line 23 based on the cost estimate we charge the tenant and it is updated during Refresh; for each new measurement of request r_f^j, if we still have credit, we subtract from it (line 12), otherwise we increase tenant's clock forward (line 14). After completion of request, we reconcile the final resource measurement with the remaining credit on line 31. We update the per-tenant L_{max}^f estimate on line 30, and use it for eligibility and finish time calculation on lines 20 and 24.

of the burst is proportional to c_r – the longer the request, the longer it will take to incorporate the true cost back in the scheduler and the longer other tenants will be blocked.

We are less concerned with the other extreme when a tenant's cost estimate is high and it transitions to cheap requests; *i.e.* $l_r \gg c_r$. If the scheduler mistakenly estimates a cheap request to be expensive, then the scheduler might delay the request for longer than normal. However, once it is scheduled the request will not occupy a worker thread for any longer than the scheduler anticipated, so the request will not cause burstiness to other tenants. The effect will only persist for a short duration since the cheap requests complete quickly. Retroactive charging will refund the unused cost to the tenant thereby guaranteeing that it will receive its fair share of service in the long run. Most importantly, overestimation does not cause prolonged thread pool blocking for other tenants.

With known costs (§4) 2DFQ spreads requests across threads by size, a property we take advantage of to handle unpredictable costs. Since workloads contain a mix of predictable and unpredictable tenants, we limit the impact of unpredictable tenants by treating them as if they are expensive. We bias unpredictable tenants towards low-index threads to keep them away from small requests on high-index threads. 2DFQ^E uses a pessimistic cost estimator that overestimates costs as follows: individually for each tenant on each API, it tracks the cost of the largest request, L_{max}^i; after receiving the true cost measurement c_r of a just-completed request, if $c_r > L_{max}^i$, we set $L_{max}^i = c_r$, otherwise we set $L_{max}^i = \alpha L_{max}^i$, where $\alpha < 1$, but close to 1. In effect, 2DFQ^E penalizes unpredictable tenants that repeatedly vary between cheap and expensive requests by just treating all requests as expensive. Requests will run on low-index threads mixed in among expensive requests. They will not run on the high-index threads and thereby not interfere with other tenants' small requests. On the other hand, stable tenants with cheap requests will maintain lower estimates and remain on the high-index threads. The α parameter allows us to tune the trade-off between how aggressively we separate predictable tenants from unpredictable ones, and how much leeway a tenant has to send the occasional expensive request.

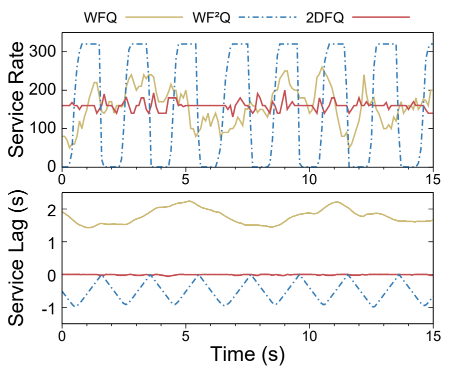
Bookkeeping: Refresh Charging When a tenant submits cheap requests for a sustained period of time, 2DFQ^E will be susceptible to underestimation if the tenant transitions to expensive requests. When this happens, L_{max}^i will be a low value that underestimates the expensive requests until their cost can be incorporated into the estimator. As we outlined in §3.2, up to N underestimated requests can run concurrently due to this large feedback delay in updating the estimator.

Refresh charging is a 'damage control' mechanism that periodically measures the resource usage of long-running requests and incorporates measurements into the scheduler immediately. It lets us notice expensive requests quickly and immediately charge the tenant for the excess cost while the request is still running, short-circuiting the typical cost estimation feedback loop. The computational overhead for measuring resource consumption is non-negligible, so the scheduler must strike a balance between frequent measurements and acceptable overheads. In practice we found that refresh charging every 10ms had no significant overhead.

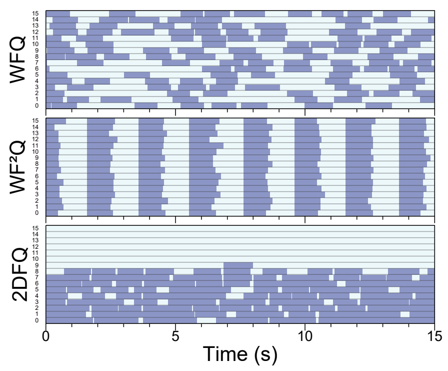
Algorithm We summarize 2DFQ^E in Figure 7.

6. EVALUATION

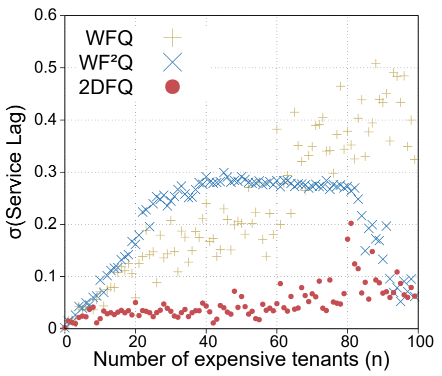
In this section, we evaluate the following goals: that 2DFQ provides smooth service to all tenants when request costs are known, despite the presence of expensive requests; and that 2DFQ^E provides smooth service to predictable tenants when request costs are unknown, despite the presence of unpredictable requests.



(a) Top: service received by a small tenant over time, measured in 100ms intervals. Bottom: service lag in seconds compared to an ideal GPS server.



(b) Thread occupancy over time. Horizontal lines represent worker threads; shaded indicates the worker is processing an expensive request, unshaded indicates a cheap request.



(c) Standard deviation (in seconds) of a small tenant's service lag in the presence of increasingly many expensive tenants. Note that small tenant's request execution time is 0.001s.

Figure 8: Synthetic workload described in §6.1.1.

We implemented all schedulers in a discrete event simulator where requests were scheduled across a fixed number of threads. We used synthetic workloads and traces from Azure Storage [4, 9] to keep the server busy throughout the experiments, but also ran experiments at lower utilizations.

We compare 2DFQ to WFQ and WF^{2}Q as baseline algorithms. Besides WFQ and WF^{2}Q, we implemented several other algorithms including SFQ [23], MSF^{2}Q [8], and DRR [50]. However, we omit these algorithms from our evaluation as we found the results to be visually indistinguishable from either WFQ or WF^{2}Q– occurring because the key differences between the algorithms are incidental to their fairness bounds. For example, since we do not use a variable rate server, the primary feature of SFQ is not necessary and SFQ and WFQ produced nearly identical schedules. Similarly, WF^{2}Q and MSF ^{2}Q produced nearly identical results; MSF^{2}Q's distinguishing feature handles the case where one tenant has a high weight or few tenants are sharing many links, whereas we evaluate with many tenants (up to several hundred) and equal weights. Furthermore, many algorithms such as DRR [50] and WF ^{2}Q$^+$ [5] improve algorithmic complexity but do not improve fairness bounds or add additional features; in practice they have similar or worse behavior compared to WFQ or WF^{2}Q. To evaluate the schedulers, we use the following metrics:

Service lag: the difference between the service a tenant should have received under GPS and the actual work done. For N threads with r processing rate, we use a reference GPS system with rate Nr.

Service lag variation: the standard deviation, σ of service lag. Bursty schedules have high service lag variation due to oscillations in service.

Service rate: work done measured in 100ms intervals.

Latency: time between the request being enqueued and finishing processing. We focus on the 99th percentile of latency, unless otherwise noted.

Gini index: an instantaneous measure of scheduler fairness across all tenants [49].

Evaluation summary Our evaluation of 2DFQ shows that:

- When request costs are known, for both synthetic (§6.1.1) and real-world (§6.1.2) workloads, 2DFQ provides service to small and medium tenants that has one to two orders of magnitude reduction in service lag variation.
- When many tenants have expensive requests, 2DFQ maintains low service lag variation for small tenants (§6.1.1).
- When request costs are unknown, 2DFQ^E reduces the service lag variation by one to two orders of magnitude for small and medium tenants (§6.2.2).
- With increasingly unpredictable workloads, 2DFQ^E improves tail latency of predictable tenants by up to 100× (§6.2.1).
- Across a suite of experiments based on production workloads, 2DFQ^E improves 99th percentile latency for predictable tenants by up to 198× (§6.2.2)

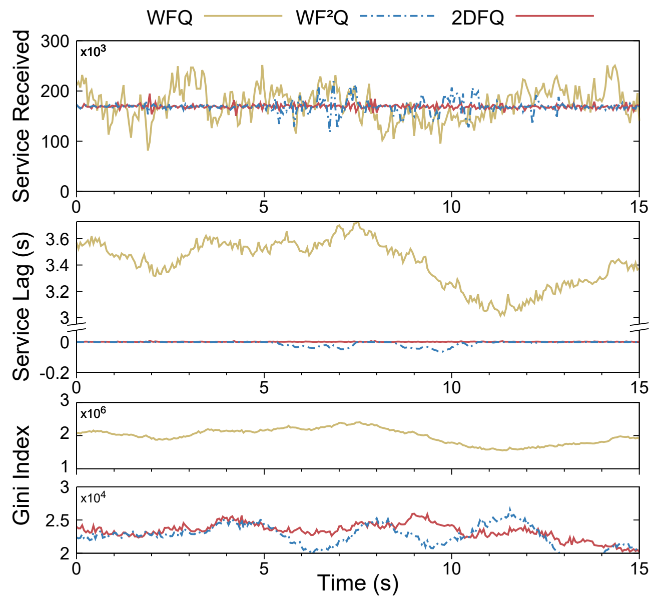
6.1 Known Request Costs

Our first set of experiments focuses on scheduling with known request costs that may vary by several orders of magnitude. We first evaluate 2DFQ under workloads with increasingly many expensive requests, and compare with the service provided under WFQ and WF^{2}Q. Second, we evaluate the overall service and fairness provided by 2DFQ on a workload derived from production traces.

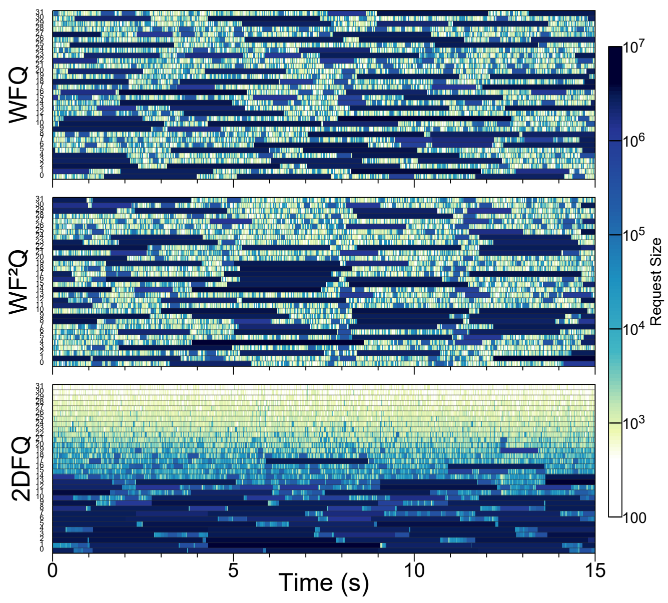
6.1.1 Expensive Requests

In this experiment we simulate the service received by 100 backlogged tenants sharing a server with 16 worker threads, each with a capacity of 1000 units per second. For varying values of n, we designate n of the tenants as small and $100 - n$ of the tenants as expensive. Small tenants sample request sizes from a normal distribution with mean 1, standard deviation 0.1; large tenants sample request sizes from a normal distribution with mean 1000, standard deviation 100.

Figure 8a examines the service received over a 15 second interval for one of the small tenants, T, when 50% of tenants are expensive ($n = 50$). Since the thread pool has 16 threads, the ideal schedule would split cheap and expensive requests into separate threads, producing steady service of 160 units per second per tenant. Figure 8a (top) shows that the service provided by WFQ has large-scale oscillations. This occurs because WFQ alternates between phases of servicing all of the 50 small tenants, followed by all of the 50 large tenants, in bursts of up to 1 thousand units per tenant. Figure 8a (bottom)



(a) Top: service received by T_1, a tenant outlined in §3.2, while replaying traces from a production server with 32 threads. Middle: service lag for the same tenant, compared to an ideal GPS server; red horizontal line is 2DFQ, which has very little service lag. Bottom: Gini index [49] across all tenants.



(b) Illustration of request sizes running on each thread. Horizontal lines represent worker threads; fill color indicates the running request size at each instant in time. WF²Q service oscillations can be identified in the middle plot by dark-colored blocks of expensive requests. 2DFQ partitions requests by size.

Figure 9: Time series for a production workload on a server with 32 threads

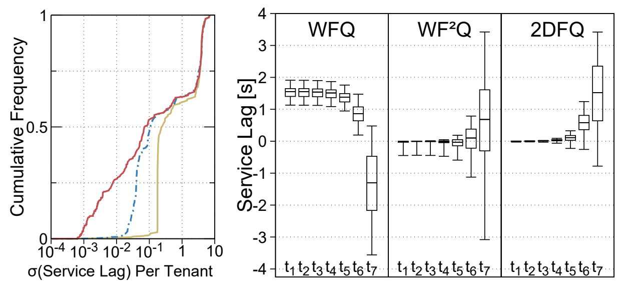


Figure 10: Left: CDF of service lag standard deviation across all tenants. 2DFQ reduces service lag standard deviation for tenants with small requests. Right: Distribution of service lag experienced by $t_1 \dots t_7$. The wider the distribution of service lag, the more oscillations a tenant will experience.

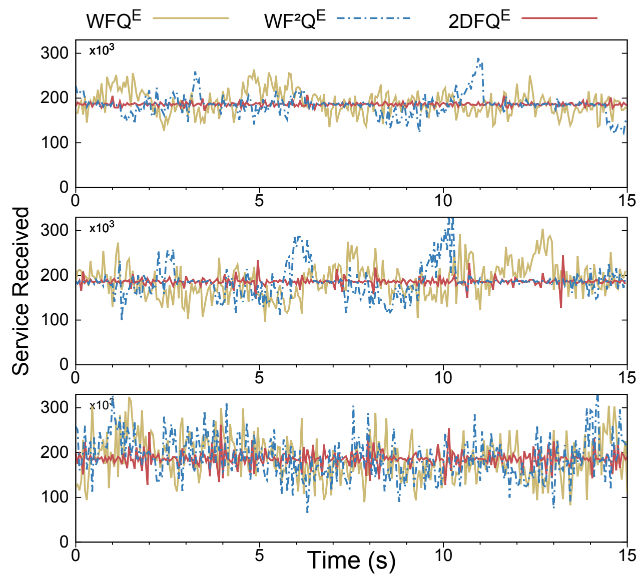
plots the service lag over time, showing that small tenants oscillate between 1 and 2 seconds ahead of their fair share, with a period of approximately 6.25 seconds. Small tenants are consistently ahead of their fair share because small requests have the earliest finish time so WFQ services them first. WF²Q has less long-term oscillation, but suffers from more extreme oscillations over shorter time scales; the small tenant receives no service for almost a second. By design, WF²Q prevents T from getting too far ahead of its fair share, but due to the presence of the expensive tenants, T continually falls behind by up to 1 second. This occurs because WF²Q determines that all small tenants are ineligible, and schedules expensive requests to run on every worker thread, as illustrated in Figure 8b. Note that because average execution time of a small tenant's request is 1ms, such rate oscillation delays the tenant by up to 1000 requests. Finally, the service provided by 2DFQ is more stable, but still with occasional oscillations. The oscillations are characterized as a period of slightly reduced service followed by a burst of increased service. As illustrated in Figure 8b (bottom), 2DFQ mostly partitions requests by size across the threads, and the remaining oscillations are a side effect of randomness in request sizes that enables expensive requests to temporarily run on 9 of the worker threads instead of 8.

We varied the proportion of expensive tenants n between 0 and 100 and show the resulting standard deviation of service lag in Figure 8c. WFQ and WF²Q experience a linear increase in standard deviation as the proportion of expensive tenants grows. WFQ grows unboundedly, whereas WF²Q eventually plateaus. With only 25% of the workload comprising expensive tenants, WF²Q converges to its worst-case behavior. On the other hand, while 2DFQ also sees gradually increased standard deviation, it is an order of magnitude lower compared to other schedulers.

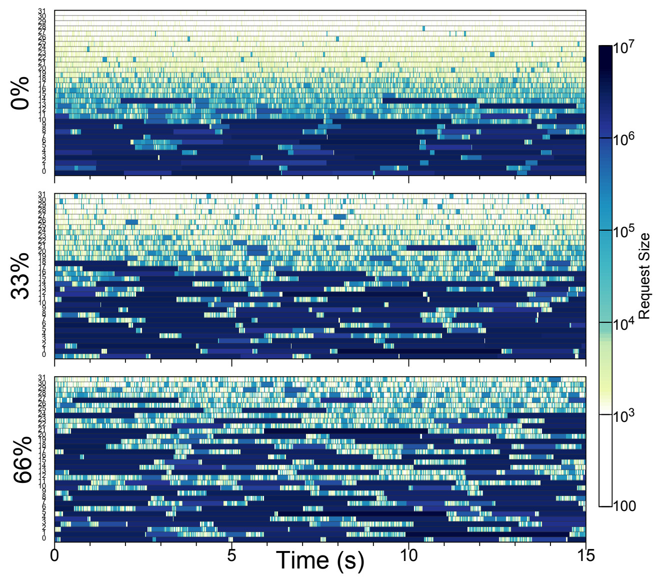
6.1.2 Production Workloads

In this experiment we evaluate fair share provided by 2DFQ with a workload derived from production traces of Azure Storage. We simulate the service received by tenants sharing a server of 32 worker threads, each with capacity 1 million units. We replay 250 randomly chosen tenants drawn from workload traces of 50 servers. As a baseline for evaluation in this and subsequent experiments, we include tenants $T_1 \dots T_{12}$ described in §3.2. In aggregate across all tenants, request costs for this experiment vary from 250 to 5 million.

We first illustrate the improved service for tenants with small requests. Figure 9a (top) shows a 15 second time series for T_1, comprising primarily small requests between 250 and 1000 in size. Figure 9a (middle) plots T_1's service lag. Under WFQ, the service received oscillates between 3s and 3.7s ahead of GPS. WF²Q more closely matches GPS, but occasionally falls behind by up to 50ms due to the thread pool becoming occupied by expensive requests. 2DFQ (the horizontal red line) closely matches GPS at all times. Figure 9a (bottom) plots the Gini index [49] over time, an aggregate measure of fairness across all tenants. WFQ is significantly less fair in



(a) Service received by T_1, a predictable tenant submitting small requests. Each figure illustrates the service received by T_1 under WFQ^E, WF^{2}Q^E, and 2DFQ^E. 2DFQ continues to provide stable service as the workload mix becomes unpredictable, with only a minor increase in oscillations.



(b) Thread occupancy over time for 2DFQ^E for increasingly unpredictable workloads. Horizontal lines represent worker threads; fill color indicates cost of the running request at each instant in time. 2DFQ^E isolates predictable, small requests even as increasingly many tenants are unpredictable.

Figure 11: Time series as the overall server workload mix comprises 0% (top), 33% (middle), and 66% (bottom) unpredictable tenants.

aggregate, while 2DFQ and WF^{2}Q are comparable. Figure 9b illustrates sizes of requests running on threads during the experiment. Service spikes under WF^{2}Q correlate with several large requests occupying threads simultaneously. 2DFQ partitions requests across threads according to request size, avoiding such spikes. Figure 10 plots a CDF of the service lag standard deviations across all tenants included in the experiment. A low standard deviation is desirable, as it corresponds to fewer oscillations in service. The figure shows that the first quartile of tenants have approximately 50x lower standard deviation under 2DFQ than WF^{2}Q and 100x lower standard deviation under 2DFQ than WFQ. These tenants are the ones with primarily small requests.

To more precisely understand how 2DFQ impacts tenants based on request sizes, we repeat the experiment and include an additional seven tenants, $t_1 \dots t_7$. These tenants submit requests with fixed costs of $2^8, 2^{10}, 2^{12}, \dots, 2^{20}$ respectively (from 256 to 1 million), spanning the range of costs in our workload. Figure 10 (right) plots the distribution of service lag experienced by $t_1 \dots t_7$ under WFQ, WF^{2}Q and 2DFQ. Each distribution shows how much the tenant deviates from its fair share. Under all schedulers, large requests (t_7) experience a wide range of service lag, because service is received in large, coarse-grained bursts. For progressively smaller requests ($t_6 \dots t_1$), WFQ reduces service lag to a range of 0.8 seconds; WF^{2}Q reduces it to 0.5 seconds, while 2DFQ reduces it to 0.01 seconds. These results illustrate how 2DFQ particularly improves the service received by tenants with small requests.

6.2 Unknown Request Costs

Our second set of experiments evaluates schedulers when request costs are not known a priori. We compare 2DFQ^E ($\alpha = 0.99$) to variants of WFQ and WF^{2}Q that estimate request costs using per-tenant per-API exponential moving averages ($\alpha = 0.99$). We refer to them, respectively, as WFQ^E and WF^{2}Q^E. We also implemented both retroactive charging and refresh charging for WFQ^E and WF^{2}Q^E. Without these techniques, we found that the quality of schedules deteriorated by a surprising amount. It turned out to be relatively common for workloads to have back-to-back requests that differ by several orders of magnitude; without retroactive charging it takes too long to incorporate measurements back into the moving average to rectify estimation error. For the same reason, without refresh charging it would quickly lead to multiple large requests taking over the thread pool. Since the bookkeeping techniques are straightforward to implement, we applied them to all algorithms, and our experiment results only reflect the differences between scheduling logic and estimation strategy.

We first evaluate 2DFQ^E for workloads where an increasing proportion of the tenants are *unpredictable*, comparing to service under WFQ^E and WF^{2}Q^E. Second, we compare the schedulers across a suite of 150 workloads generated from production workload traces, and assess the overall effect on service lag and request latency.

6.2.1 *Unpredictable Workloads*

In this experiment we evaluate 2DFQ^E's pessimistic cost estimation strategy, demonstrating how it co-locates unpredictable and expensive tenants, keeping them away from predictable tenants with small requests. We examine a single workload in detail; in §6.2.2 we give aggregate results across a suite of experiments. We show that 2DFQ^E improves service for those tenants compared to WFQ^E and WF^{2}Q^E, which deteriorate under the same conditions.

We examine a workload of 300 randomly selected tenants plus $T_1 \dots T_{12}$ as in §6.1.2. We repeat the experiment three times. Initially, most tenants in the workload are predictable, and 2DFQ^E provides little, if any, improvement over WFQ^E

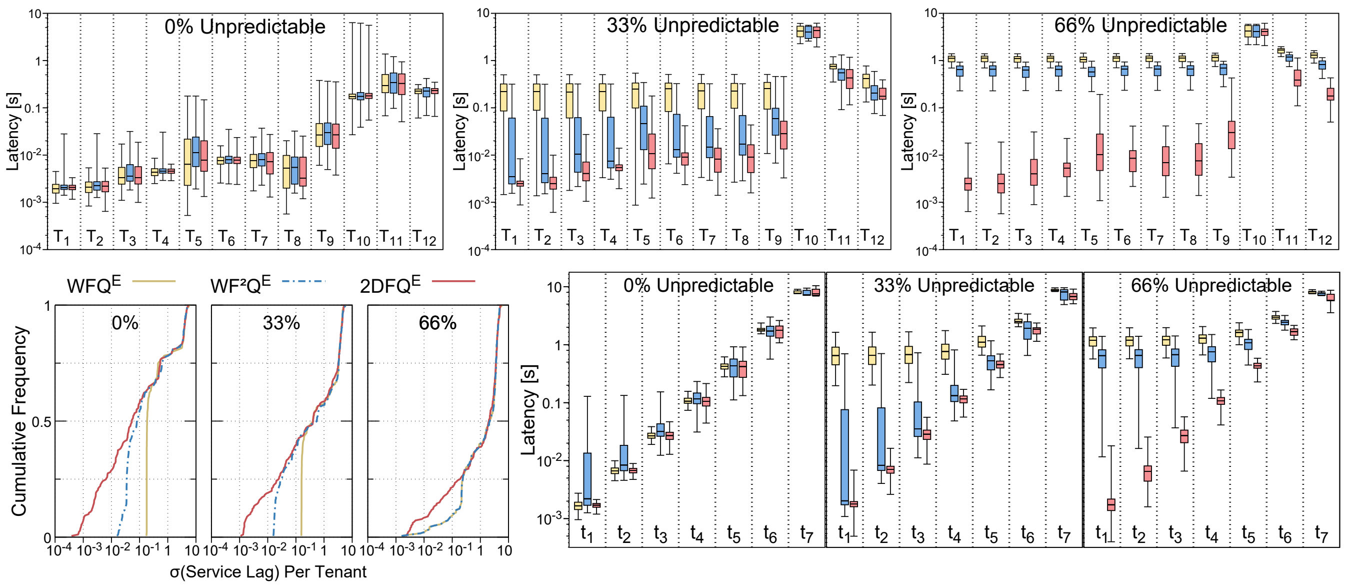


Figure 12: Request latencies as the overall server workload mix is increasingly unpredictable. Top row: latency distributions for $T_1 \ldots T_{12}$ with 1% and 99% whiskers; each cluster of three bars shows the latency distribution for one tenant under WFQ^E, WF^2Q^E, and $2DFQ^E$ respectively. $2DFQ^E$ mitigates the impact of unpredictable tenants and significantly improves latencies for predictable tenants with small requests, such as T_1. Bottom left: CDFs of service lag standard deviation. Bottom right: latency boxplots for fixed-cost tenants $t_1 \ldots t_7$.

and WF^2Q^E. However, for the second and third repetitions of the experiment, we make 33% and 66% of these tenants explicitly unpredictable, by sampling each request pseudo-randomly from across all production traces disregarding the originating server or account. The resulting tenants lack predictability in API type and cost that is common to real-world tenants as shown in §3.2, and as the workload becomes unpredictable, WFQ^E and WF^2Q^E rapidly deteriorate.

Figure 11a plots a time series of the service received by T_1 under WFQ^E, WF^2Q^E, and $2DFQ^E$. The top figure shows that with the baseline workload, WFQ^E provides service with the most oscillations; WF^2Q^E provides service with occasional spikes, and $2DFQ^E$ provides consistently smooth service. Oscillations under WFQ^E are lower than they were in experiment 6.1.2 since a side effect of EMA averaging is to make costs more uniform across tenants: for tenants with any variation in request size, small requests are perceived to be larger than they are and large requests are perceived to be smaller. The middle and bottom figures demonstrate how the service deteriorates with increasingly unpredictable workloads (33% middle; 66% bottom). WFQ^E and WF^2Q^E produce large scale oscillations in service, while $2DFQ^E$ has occasional spikes of service.

Figure 11b illustrates the schedules produced by $2DFQ^E$ for the three experiments. The figure shows how $2DFQ^E$ initially partitions requests accurately according to cost, but as more unpredictable tenants are present, the partitioning becomes more coarse grained. This occurs when a request is estimated to be large but it is small, or vice versa. It can be observed by small requests interspersed among large requests, and vice versa. Each of the brief spikes in service experienced by T_1 in Figure 11a can be correlated with an temporary imbalance of expensive requests.

Oscillations in service can have a profound effect on request latencies. Figure 12 shows boxplot latency distributions for $T_1 \ldots T_{12}$ (top row), with whiskers highlighting 1st and 99th percentile latencies. Each cluster of three bars shows the latency distribution under WFQ^E, WF^2Q^E, and $2DFQ^E$ respectively for one tenant in one experiment. For the baseline workload (0% Unpredictable) median and tail latencies under WFQ^E and $2DFQ^E$ are comparable. WF^2Q^E has similar median latencies, but higher 99th percentile latencies for small requests because of the blocking effects outlined in §2.

However, with 33% and 66% unpredictable tenants present, latencies under WFQ^E and WF^2Q^E increase nearly uniformly across all tenants, to approximately 1 second 99th percentile latency. The relative increase in latency is most noticeable for tenants with smaller requests ($T_1 \ldots T_4$), with median and 99th percentile latencies increasing by a factor of more than 100×. By contrast with $2DFQ^E$ these tenants are significantly less impacted by the unpredictable tenants, with 99th percentile latencies increasing by a maximum factor of 10× among experiments. With 66% of the workload unpredictable, $2DFQ^E$ provides a 99th percentile latency speedup of up to 100× over WFQ^E for tenants such as T_1 with small, predictable requests. On the other hand, tenants such as T_{10} do not experience significant latency improvements. Recall Figure 4c from §2: T_{10}'s requests vary widely in cost, by more than 3 orders of magnitude. $2DFQ^E$ does not improve T_{10}'s service because it is an example of the expensive and unpredictable tenants that must be isolated from others.

Despite some tenants being more predictable than others, $T_1 \ldots T_{12}$ nonetheless have variation in request costs. To more precisely understand how $2DFQ^E$ affects latencies for tenants based on request size, we repeat the experiment to include the fixed-cost tenants $t_1 \ldots t_7$ as described in §6.1.2. Figure 12 (bottom right) shows boxplot latency distributions for $t_1 \ldots t_7$, and illustrates how the relative latency degradation disproportionately affect tenants $t_1 \ldots t_4$, whose requests are the smallest. Across all tenants, as the workload becomes less predictable, latencies converge towards the latency of the most expensive requests in the system.

Figure 12 (bottom left) plots CDFs of the service lag standard deviation across all tenants. It shows the successive increase in the proportion of tenants with high standard deviation – this corresponds to the unpredictable tenants. The remaining predictable tenants experience approximately 10 to 15× reduced standard deviation under 2DFQ^E compared to WFQ^E and WF^{2}Q^E.

6.2.2 *Production Workloads*

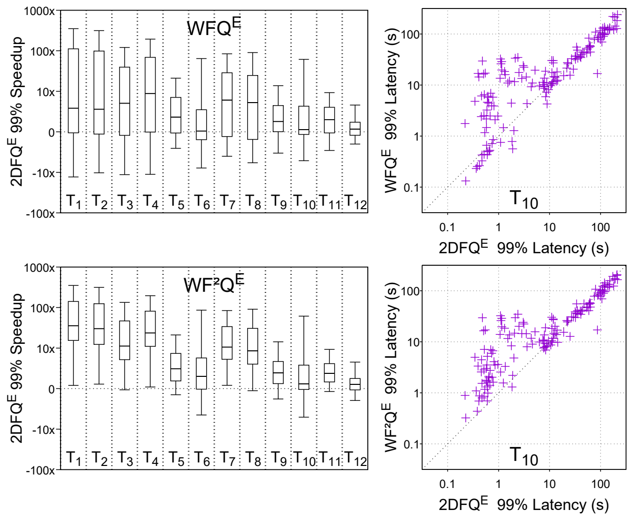
Finally, we run a suite of 150 experiments derived from production workloads of Azure Storage. We simulate the service received by tenants under WFQ^E, WF^{2}Q^E, and 2DFQ^E, as we randomly vary several parameters: the number of worker threads (2 to 64); the number of tenants to replay (0 to 400); the replay speed (0.5-4×); the number of continuously backlogged tenants (0 to 100); the number of artificially expensive tenants (0 to 100); and the number of unpredictable tenants (0 to 100). To compare between experiments, we also include $T_1 \ldots T_{12}$.

We measure the 99th percentile latency of tenants in each experiment and calculate the relative speedup of 2DFQ^E compared to WFQ^E and WF^{2}Q^E. As an example, in the "0% Unpredictable" experiment of §6.2.1, T_1's 99th percentile latency was 3.3ms under 2DFQ^E, 4.5ms under WFQ^E, and 28ms under WF^{2}Q^E, giving 2DFQ^E a speedup of 1.4× over WFQ^E and 8.5× over WF^{2}Q^E.

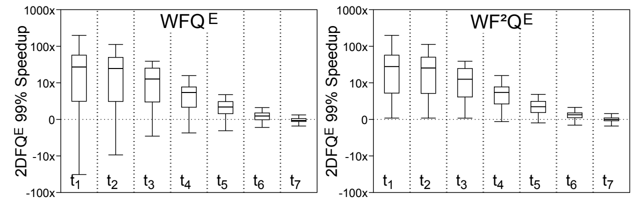
Figure 13a (left) plots the distribution of 2DFQ^E's speedup over WFQ^E and WF^{2}Q^E. Across the experiments, 2DFQ^E significantly improves 99th percentile latency for tenants such as T_1, whose requests are small and predictable (illustrated in §2). T_1 has a median improvement of 3.8× over WFQ^E and 142× over WF^{2}Q^E. However, 2DFQ^E does not improve 99th percentile latency as much for tenants with large and/or unpredictable requests, such as T_{10} and T_{12}. Figure 13a (right) plots the 99th percentile latencies across all experiments for T_{10}, comparing 2DFQ^E to WFQ^E (top) and WF^{2}Q^E (bottom). 2DFQ^E resulted in worse 99th percentile latency for T_{10} in 64 of the experiments for WFQ^E and 47 for WF^{2}Q^E. However, when 2DFQ^E did improve latencies for T_{10}, it was by significantly larger factors (up to 61×) than when latencies were worse (up to 5×).

To better understand how 2DFQ^E improves latencies for tenants with smaller requests, we repeat the experiment suite to include the fixed-cost tenants $t_1 \ldots t_7$ as described in §6.1.2. Figure 13b plots the distribution of 2DFQ^E's speedup over WFQ^E and WF^{2}Q^E for $t_1 \ldots t_7$, and illustrates how latency is primarily improved for tenants with small requests (t_1). Conversely, tenants with very expensive requests such as t_7 and T_{12} see little, if any improvement.

Overall, the experiments where WFQ^E and WFQ^E performed best correlated with low numbers of both unpredictable and expensive tenants, for which request cost estimates were accurate and there was little chance for thread pool blocking. On the other hand, the experiments where 2DFQ^E performed best correlated with high numbers of either unpredictable or expensive tenants, with most speedups occurring when



(a) 99th percentile latency speedups for $T_1 \ldots T_{12}$ comparing 2DFQ^E to WFQ^E (top row) and WF^{2}Q^E (bottom row). Scatter plots show T_{10}'s 99th percentile latencies in detail, each point representing one experiment.



(b) Results for $t_1 \ldots t_7$ from repeated experiment suite.

Figure 13: Comparison of 99th percentile latencies across 150 experiments based on production workloads. Whiskers extend to min and max values.

both were true. WFQ^E and WF^{2}Q^E rapidly deteriorate as the workload becomes unpredictable, increasing the relative improvement of 2DFQ^E. For example, in §6.2.1, the initial workload (0% Unpredictable) was more predictable than a typical workload, and 2DFQ only improved 99th percentile latency for T_1 by a factor of approximately 1.5× over WFQ^E. On the other hand, the final workload (66% Unpredictable) was less predictable than a typical workload, and T_1's 99th percentile latency improvement over WFQ^E was more than 100×. The suite of 150 experiments presented here similarly vary in how predictable they are, and the range of 99th percentile latency speedups illustrated in Figure 13a (left) reflect this range.

7. DISCUSSION

2DFQ Intuition 2DFQ improves service compared to WFQ and WF^{2}Q primarily due to the manageable mix of predictable, unpredictable, cheap and expensive requests in real workloads. Consider the illustration in Figure 14. At one extreme we have predictable workloads with low variation in request cost (①). This scenario is representative of packet scheduling on network links, and WFQ, WF^{2}Q and 2DFQ would provide similarly good quality of service because little, if any, blocking can occur. At the other extreme lies workloads with hugely varying request costs and completely unpredictable tenants (②). In this scenario, all schedulers would behave poorly because blocking would be unavoidable, even for 2DFQ. However, starting from

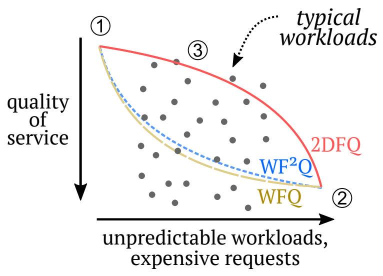


Figure 14: Illustration of the intuition behind 2DFQ's significantly improved service for our workloads.

the first extreme and transitioning to the second, with workloads becoming less predictable, 2DFQ does not deteriorate as rapidly as WFQ or WF^{2}Q do. Between the two extremes lies a middle ground where WFQ and WF^{2}Q experience blocking and reduced quality of service, but 2DFQ does not (③). Many real-world workloads lie between these two extremes, containing both unpredictable and predictable tenants. Our results in §6.2.1 and §6.1.1 demonstrated this deterioration.

Estimators We designed 2DFQ^E's pessimistic estimation strategy to take advantage of 2DFQ's cost-based partitioning. WFQ and WF^{2}Q lack cost-based partitioning, so there is nothing fundamental about them that would benefit from this estimation strategy. Nonetheless, we can apply it to these algorithms; we experimented with numerous combinations of scheduler and estimator, and found that WFQ and WF^{2}Q with pessimistic estimation performed no better, and often significantly worse, than using an EMA. We view estimator choice as an important design point for future work in this space – to ensure good behavior when over- or under-estimating request costs.

Limitations While 2DFQ improves quality of service when the system is backlogged, work-conserving schedulers in general cannot improve service when the system is under-utilized. Inevitably, all worker threads could be servicing expensive requests if no other requests are present. Any subsequent burst of small requests would have to wait for the expensive requests to finish. This behavior occurs under 2DFQ and all non-preemptive schedulers, and creates large delay for the small requests. One way to avoid this is to make the scheduler not work-conserving, for example, by allowing threads to remain idle despite the presence of queued requests. Another option is to allow a variable number of worker threads and to spawn new threads when small requests show up. This would over-saturate the CPU and thus slow down already running requests, but would allow the small requests to finish faster; however, it would incur additional overhead from more context-switching. In the extreme, we could take a thread-per-tenant approach; however, this results in more context switching, contention for application level (*e.g.*, locks, caches) and system level (*e.g.*, disk) resources, and substantially reduced goodput. This is especially relevant since requests can be very short – less than 1ms in duration for many requests – which exacerbates context switching overheads. Our approach in this paper – fair queue scheduling at the application level – is the preferred approach in cloud services [15] for efficiency and to keep low-level resource queues (*e.g.*, disk) short.

The challenges presented in this paper are the result of wide cost variation in cloud services. An alternative approach is to reduce cost variation by splitting up long requests into shorter ones [15]. For example, after 100ms of work a request could pause and re-enter the scheduler queue. However, this approach implies more overhead on the developer; it can be applied only to certain request types; and it affects execution efficiency because of, *e.g.*, data loaded in various system caches.

8. RELATED WORK

We discuss related work based on the core challenges addressed in this paper.

Packet Scheduling Fair queue scheduling has been developed in the context of packet scheduling [6,23,46,50], where packets are sent sequentially across a single link. Because of this, the majority of papers in this area only consider sequential execution. An exception is, for example, MSF^{2}Q [8], where the authors consider scheduling packets across multiple aggregated links. While this setting is very close to ours, the scheduler is a direct extension of WF^{2}Q and we found that it produces near identical behavior in the presence of large requests. Single-link schedulers such as WFQ [46] or WF^{2}Q [6] require packet size to schedule because they order flows based on their finish tag. SFQ [23] does not need the packet size before scheduling the packet because it selects packets based on their start tag, which is computed based on sizes of previous packets. When a packet completes, its observed cost is used to update the start tag of its flow. When applied to request scheduling where multiple requests of the same tenant can execute concurrently, this approach, however, does not work because we would have to execute each tenant sequentially.

Thread Scheduling and Pre-Emption Thread scheduling in the operating system is analogous to packet scheduling. In the research literature, lottery scheduling [58] and stride scheduling [59] were independently developed and later found to be equivalent to fair queuing [17,46]. Scheduling algorithms have proliferated in both domains, for example start-time fair queuing was proposed for both hierarchical link [23] and CPU [22] sharing. On multicore systems, hierarchical schedulers such as the Linux Completely Fair Scheduler [45] and Distributed Weighted Round Robin [40] extend fair queuing to multiple cores by maintaining per-core run queues and load-balancing runnable threads across cores.

Thread schedulers can control the amount of time a thread spends running on a core (*i.e.*, the quantum or time slice). Thus they have the means to explicitly bound how long a core can be occupied before a different thread gets to run. In the worst case, this reduces burstiness to the granularity of the largest time slice. For example, Li *et al.* [40] discuss for infeasible thread weights: "*Eventually, this thread becomes the only one on its CPU, which is the best any design can do to fulfill an infeasible weight.*"

However, applications cannot control preemption or specify fairness goals because operating systems do not expose sufficient control over these mechanisms. Current operating systems do not give applications the ability to configure thread preemption; at most, Windows User-Mode Scheduling [55] (most notably used by Microsoft SQL Server [35]) gives applications control over thread scheduling but lacks configurable time slices – threads only yield to the scheduler when they make blocking systems calls or a direct call to `UmsThreadYield()`. Operating systems also lack application-level tenant information and do not have access to application-level request queues. Fairness mechanisms like cgroups [11] enable operators to divide resources between processes and threads to provide fairness and isolation, but do not have access to application-level queues. More importantly, due to the high number of tenants, a thread-per-tenant approach is infeasible; short requests less than 1ms in execution duration exacerbate context-switching overheads and reduce throughput, while higher concurrency increases contention over application-level (*e.g.*, locks, caches) and system-level (*e.g.*, disk) resources.

Event-based systems have long been debated in the operating systems community as a dual to thread-based systems [39,44,56]. A key feature of event-based systems is *cooperative multitasking*: event handlers are not preemptible and run until completion, simplifying concurrent programming on single-core machines because event handlers are implicitly atomic [44,56]. Thread-based systems also adapted this feature into cooperative scheduling [1,57], whereby threads only yield to the scheduler at pre-defined points specified by the developer. For both event-based and thread-based systems, cooperative scheduling is vulnerable to long-running event handlers, or threads that go for a long time without yielding to the scheduler. When this occurs, programs can block for large periods of time and the program may become non-responsive [1,44,57]. To avoid this behavior, developers can explicitly split up long-running threads or handlers into smaller ones that reenter the scheduler more frequently. This solution is similarly applicable in our domain, and is the approach taken, for example, by Google's Web search system [15]. However, it requires manual intervention from developers, and only reduces the range of request costs – it does not eliminate variation entirely. The approach is fundamentally constrained by factors that affect execution efficiency, *e.g.* data loaded in various system caches and intermediary memory allocation, and is burdened by need for "stack ripping" [1]. An alternative to manual intervention is framework support for automatically reentering the scheduler, for example by analyzing code to identify the boundaries of critical sections [57], or as part of the language runtime [18]. In all of these systems, if fairness is a goal, then 2DFQ can be used to provide smooth average-case schedules.

Middlebox Packet Processing Dominant-Resource Fair Queuing (DRFQ) [21], a multi-resource queue scheduler for middlebox packet processing, allows concurrent execution of multiple requests, such as on the CPU, but does not deal with large variation in request costs and only permits serial execution for each tenant. DRFQ builds on top of SFQ and uses linear resource consumption models for different types of requests. The authors show that for several middle-box modules linear models work relatively well, but acknowledge that if models are inaccurate, allocated shares might be off proportionally to the estimation error. Further, because the resource models depend on which modules the packet executed in, resource accounting happens only *after* the request completes, which limits DRFQ to executing single tenant's packets sequentially.

Storage and I/O pClock [26], mClock [25], and Pisces [52] propose queue schedulers for physical storage, where several I/O requests execute concurrently. I/O request costs are much less variable than in the cloud setting, and dynamic workloads remain an open challenge [61]. Similar request cost modeling has been done in the storage domain as well [25,54], where type of operations and hardware variability are limited. For example, IOFlow [54] periodically benchmarks the storage device to estimate costs of tokens used for pacing requests. Also, to bound the uncertainty of arbitrary long IO requests, they break them into 1MB requests.

Distributed Systems Many distributed systems schedulers, such as Retro [43], Cake [60], and Pulsar [3] periodically measure request costs and use these estimates in the next interval. However, in dynamic workloads, such as shown in Figure 4, such approach can lead to arbitrary unfairness across tenants unless estimation errors are addressed. These systems enforce fair share using rate limiters, typically implemented as token buckets, which are not designed to provide fairness at short time intervals. Depending on the token bucket rate and burst parameters, they can either under-utilize the system or concurrent bursts can overload it without providing any further fairness guarantees.

Web Applications A large body of work – for example [7, 48] or see [24] for a survey – has focused on providing *differentiated services* or *quality-of-service (QoS)* for cluster-based applications; they define multiple user *classes* (or tenants) with different scheduling policies based on priorities, achieved utility or required resources. These papers typically consider problems related to admission control, allocating resources to maximize total utility, or distributed scheduling and do not deal with providing fine-grained resource fairness. Scheduling requests with inaccurate or unknown size has been studied previously [2,62]. However, these papers concentrate on various priority-based policies, such as shortest-job-first or shortest-remaining-time-first, and ignore resource fairness. For example, Aalo [12], schedules *co-flows* in a network without prior knowledge of their size; by using a priority queue where new flows start at the highest priority and their priority decreases as they send more data.

9. CONCLUSION

In this paper we demonstrated the challenges of fair queuing in multi-tenant services, where requests with large cost variance execute concurrently across many threads. We proposed and evaluated a practical scheduler for such settings, Two-Dimensional Fair Queuing, which achieves significantly more smooth schedules and can improve latencies of small requests when competing with large requests.

10. REFERENCES

[1] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative task management without manual stack management. *2002 USENIX Annual Technical Conference (ATC '02)*. (§8).

[2] Amico, M. D., Carra, D., and Michiardi, P. PSBS: Practical size-based scheduling. *IEEE Transactions on Computers* (2016). (§8).

[3] Angel, S., Ballani, H., Karagiannis, T., O'Shea, G., and Thereska, E. End-to-end performance isolation through virtual datacenters. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§2, 3.2, and 8).

[4] Microsoft Azure Storage. https://azure.microsoft.com/services/storage/. [Online; accessed June 2016]. (§1, 3, and 6).

[5] Bennett, J. C., and Zhang, H. Hierarchical packet fair queueing algorithms. *1996 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '96)*. (§6).

[6] Bennett, J. C., and Zhang, H. WF^{2}Q: Worst-case fair weighted fair queueing. *15th IEEE Conference on Computer Communications (INFOCOM '96)*. (§1, 2, 4, and 8).

[7] Blanquer, J. M., Batchelli, A., Schauser, K., and Wolski, R. Quorum: Flexible quality of service for internet services. *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*. (§8).

[8] Blanquer, J. M., and Özden, B. Fair queuing for aggregated multiple links. *2001 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '01)*. (§1, 2, 4, 6, and 8).

[9] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. (§1, 1, 2, 3, and 6).

[10] CASSANDRA-8032: User based request scheduler. https://goo.gl/PhHhai. [Online; accessed June 2016]. (§2).

[11] Linux Control Groups. https://goo.gl/DDsmig. [Online; accessed June 2016]. (§8).

[12] Chowdhury, M., and Stoica, I. Efficient coflow scheduling without prior knowledge. *2015 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '15)*. (§8).

[13] CLOUDSTACK-618: API request throttling to avoid malicious attacks on MS per account through frequent API request. https://goo.gl/m8F4Ic. [Online; accessed June 2016]. (§2).

[14] API Request Throttling. https://goo.gl/gl1bGE. [Online; accessed June 2016]. (§1 and 2).

[15] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80. (§7 and 8).

[16] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. *21st ACM Symposium on Operating Systems Principles (SOSP '07)*. (§1).

[17] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. *1989 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '89)*. (§8).

[18] Deshpande, N., Sponsler, E., and Weiss, N. Analysis of the go runtime scheduler. https://goo.gl/JRfNn3, 2012. [Online; accessed June 2016]. (§8).

[19] Summary of the Amazon DynamoDB Service Disruption. https://goo.gl/R2SKKs. [Online; accessed June 2016]. (§1).

[20] Quality of Service in Hadoop. http://goo.gl/diwR00. [Online; accessed June 2016]. (§1 and 2).

[21] Ghodsi, A., Sekar, V., Zaharia, M., and Stoica, I. Multi-resource Fair Queueing for Packet Processing. *2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '12)*. (§3.1 and 8).

[22] Goyal, P., Guo, X., and Vin, H. M. A hierarchical cpu scheduler for multimedia operating systems. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*. (§8).

[23] Goyal, P., Vin, H. M., and Chen, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *1996 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '96)*. (§2, 6, and 8).

[24] Guitart, J., Torres, J., and Ayguadé, E. A survey on performance management for internet applications. *Concurrency and Computation: Practice and Experience 22*, 1 (2010), 68–106. (§8).

[25] Gulati, A., Merchant, A., and Varman, P. J. mClock: handling throughput variability for hypervisor IO scheduling. *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. (§8).

[26] Gulati, A., Merchant, A., and Varman, P. J. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. *2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. (§8).

[27] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure Recovery: When the Cure Is Worse Than the Disease. *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS '13)*. (§1 and 2).

[28] HADOOP-3810: NameNode seems unstable on a cluster with little space left. https://goo.gl/nl2mWL. [Online; accessed June 2016]. (§1 and 2).

[29] HADOOP-9640: RPC Congestion Control with FairCallQueue. https://goo.gl/ucFHWJ. [Online; accessed June 2016]. (§2).

[30] Apache HBase. http://hbase.apache.org. [Online; accessed June 2016]. (§2).

[31] HBASE-11598: Add simple RPC throttling. https://goo.gl/mxokpa. [Online; accessed June 2016]. (§2).

[32] HDFS-4183: Throttle block recovery. https://goo.gl/EGYvuX. [Online; accessed June 2016]. (§2).

[33] HDFS-945: Make NameNode resilient to DoS attacks (malicious or otherwise). https://goo.gl/YCAYgk. [Online; accessed June 2016]. (§1 and 2).

[34] Denial of Service Resilience. https://goo.gl/SXP1wG. [Online; accessed June 2016]. (§2).

[35] Henderson, K. The guru's guide to sql server architecture and internals. (§8).

[36] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *2010 USENIX Annual Technical Conference (ATC '10)*. (§1 and 2).

[37] Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., et al. Impala: A modern, open-source SQL engine for hadoop. *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*. (§2).

[38] KUDU-1395: Scanner KeepAlive requests can get starved on an overloaded server. https://goo.gl/A5VNUO. [Online; accessed June 2016]. (§2).

[39] Lauer, H. C., and Needham, R. M. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review 13*, 2 (1979), 3–19. (§8).

[40] Li, T., Baumberger, D., and Hahn, S. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *14th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. (§8).

[41] Lipcon, T., Alves, D., Burkert, D., Cryans, J.-D., Dembo, A., Percy, M., Rus, S., Wang, D., Bertozzi, M., McCabe, C. P., and Wang, A. Kudu: Storage for Fast Analytics on Fast Data. http://getkudu.io/kudu.pdf. [Online; accessed June 2016]. (§2).

[42] Lu, H., Saltaformaggio, B., Kompella, R., and Xu, D. vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation. *6th ACM Symposium on Cloud Computing (SoCC '15)*. (§3.2).

[43] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. (§2, 3.2, and 8).

[44] Ousterhout, J. Why threads are a bad idea (for most purposes). *1996 USENIX Annual Technical Conference (ATC '96)*. [Presentation]. (§8).

[45] Pabla, C. S. Completely fair scheduler. *Linux Journal*, 184 (2009), 4. (§8).

[46] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks-the single-node case. *11th IEEE Conference on Computer Communications (INFOCOM '92)*. (§1, 2, and 8).

[47] Schuller, P. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. https://goo.gl/lcZJWC. [Online; accessed June 2016]. (§2).

[48] Shen, K., Tang, H., Yang, T., and Chu, L. Integrated resource management for cluster-based internet services. *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*. (§8).

[49] Shi, H., Sethu, H., and Kanhere, S. S. An evaluation of fair packet schedulers using a novel measure of instantaneous fairness. *Computer communications 28*, 17 (2005), 1925–1937. (§6, 9a, and 6.1.2).

[50] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. *1995 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '95)*. (§2, 6, and 8).

[51] Shue, D., and Freedman, M. J. From application requests to virtual iops: Provisioned key-value storage with libra. *9th ACM European Conference on Computer Systems (EuroSys '14)*. (§3.2).

[52] Shue, D., Freedman, M. J., and Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. (§2, 3.2, and 8).

[53] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*. (§1 and 2).

[54] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. IOFlow: A Software-defined Storage Architecture. *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. (§2 and 8).

[55] User-Mode Scheduling. https://goo.gl/0JJRNl. [Online; accessed June 2016]. (§8).

[56] Von Behren, J. R., Condit, J., and Brewer, E. A. Why events are a bad idea (for high-concurrency servers). *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS '03)*. (§8).

[57] Von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. Capriccio: scalable threads for internet services. *19th ACM Symposium on Operating Systems Principles (SOSP '03)*. (§8).

[58] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*. (§8).

[59] Waldspurger, C. A., and Weihl, W. E. Stride scheduling: Deterministic proportional share resource management. (§8).

[60] Wang, A., Venkataraman, S., Alspaugh, S., Katz, R., and Stoica, I. Cake: Enabling High-level SLOs on Shared Storage Systems. *3rd ACM Symposium on Cloud Computing (SoCC '12)*. (§2 and 8).

[61] Wang, H., and Varman, P. J. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. *12th USENIX Conference on File and Storage Technologies (FAST 14)*. (§2 and 8).

[62] Wierman, A., and Nuyens, M. Scheduling despite inexact job-size information. *2008 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*. (§8).