Sandpaper: Mitigating performance interference in CDN edge proxies

Jeffrey Helt
Princeton University
jhelt@cs.princeton.edu

Guoyao Feng
Carnegie Mellon University
gfeng@cs.cmu.edu

Srinivasan Seshan
Carnegie Mellon University
srini@cs.cmu.edu

Vyas Sekar
Carnegie Mellon University
vsekar@andrew.cmu.edu

ABSTRACT

Modern content delivery networks (CDNs) allow their customers (i.e., operators of web services) to customize the processing of requests by uploading and executing code at the edges of the CDN's network. To achieve scale, CDNs have forgone heavyweight virtualization techniques. Instead, all requests often execute within the same OS or even process. However, performance interference may arise when these requests have differing demands on multiple system resources. In this paper, we study the sources of performance interference based on workloads from real customers, identify the lack of multi-resource fairness as the culprit, and show that existing schedulers available in commodity OSs are insufficient to enforce fairness between customers.

We then design Sandpaper, a new and practical multi-resource request scheduler for mitigating performance interference in CDN edge environments. Sandpaper enforces fairness despite constraints, such as sitting within the application runtime and running atop the OS's underlying resource schedulers. By leveraging key insights about the differences between theoretical system models and real systems, Sandpaper bridges the trade-off between resource utilization and multi-resource fairness that plagues existing schedulers. We implement Sandpaper atop Varnish, an open-source CDN edge proxy, and show that it mitigates performance interference while maintaining high resource utilization and with little performance overhead.

SEC '19, November 7–9, 2019, Arlington, VA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363313

CCS CONCEPTS

• **Networks → In-network processing**;

KEYWORDS

Fair Queuing, Content Delivery Networks, Scheduling

ACM Reference Format:
Jeffrey Helt, Guoyao Feng, Srinivasan Seshan, and Vyas Sekar. 2019. Sandpaper: Mitigating performance interference in CDN edge proxies. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318216.3363313

1 INTRODUCTION

Many CDNs today offer customizable computing environments at the edges of their networks [1, 2, 9, 11, 14, 16]. Customers use these programmable environments to improve their services, for instance, by tailoring content to users.

Several of these services, such as Fastly and SFR Telecomm, are built around Varnish, an open-source HTTP reverse proxy [1, 2, 7, 14]. With Varnish, they enable their customers—services like The New York Times—to customize request processing at the edge of their networks using a high-level, domain-specific language, Varnish Configuration Language (VCL). Other CDNS, like Cloudflare Workers [16] and CloudFront's Lambda@Edge [11], offer similar capabilities.

These new multi-tenant, edge platforms serve requests for tens of thousands of customers, with peak loads of up to tens of millions of requests per second [1, 14, 20]. Multiple threads in a single process handle requests from different customers, with fine-grained sharing of hardware and software resources and without virtualization techniques [14, 16]. Using workloads based on real customers, we demonstrate that customers can experience significant performance interference when simultaneously submitting requests to these platforms.

Through analysis of request processing bottlenecks on multiple system resources, we observe that the performance interference problem stems from requests' differing demands on multiple resources. Thus, it must be solved by enforcing multi-resource fairness [37] between customers. However, in trying to apply existing multi-resource schedulers to this problem, we find that while they are theoretically sound, their use in practical systems faces many hurdles. These problems range from deployability constraints to built-in assumptions about system structure and efficient use of parallel resources, e.g., multi-core CPUs. In this work, we aim to bridge the gap between the existing theoretical literature on multi-resource schedulers [36, 68–70] and practical deployment by developing a new multi-resource scheduler for such edge proxies that both achieves high utilization of parallel resources and has provable fairness guarantees.

In this work, we present Sandpaper, a new request scheduler to mitigate performance interference in thread-based CDN edge environments. Sandpaper runs entirely in user space and does not require changes to the underlying OS. Sandpaper comprises two components, a new worker pool size manager and a new request releaser, to tackle the challenges above. The worker pool size manager manages the number of workers in the system that service client requests to allow Sandpaper to enforce fairness despite running atop Linux's thread scheduler. Further, the request releaser leverages several properties of real systems, including run-to-completion processing and blocking network I/O, to resolve the fundamental trade-off between fairness and resource utilization in theoretical multi-resource schedulers [36, 68–70], achieving both fairness and high resource utilization.

Sandpaper relaxes the theoretical guarantees of an existing scheduler [68] to enable a more practical system design. Sandpaper effectively enforces multi-resource fairness between customers' requests and provides performance isolation even in the face of dynamic request processing costs. However, Sandpaper's performance isolation currently comes at a cost; Sandpaper's overhead is between 1% and 9% for realistic customer workloads, but other work suggests it can be reduced to negligible levels [30, 71]. Further, Sandpaper is highly scalable, easily handling multiple tens of thousands of requests per second on a single machine.

In summary, this work makes the following contributions:

- We investigate and diagnose sources of performance interference in Varnish using configurations based on real customers (§3.2).
- We design Sandpaper, a new and practical multi-resource request scheduler for CDN edge environments that does not require any modifications to the underlying operating system (§5.1).
- We derive analytical results to show that Sandpaper provides formal fairness guarantees (§5.2 and §5.3).
- We implement a prototype of Sandpaper in Varnish and demonstrate its effectiveness in mitigating performance interference between customers (§6).

2 RELATED WORK

Content Delivery Networks. Content delivery networks (CDNs) emerged in the 1990s in an effort to increase web performance [53, 58, 64]. CDNs deploy servers at the edges of the Internet in order to cache static content, lower latencies for user requests, and decrease bandwidth and infrastructure requirements for web service operators [25, 31]. As web content became more dynamic, CDN edge servers evolved to dynamically generate content on demand [31, 46, 58, 64]. Over time, these capabilities evolved into programmable environments, allowing operators to modify request processing by deploying code on the CDN's servers [9, 11, 14, 16].

Operator code was originally written in restricted programming languages [7, 10]. However, IoT and mobile computing pushed this trend further as arbitrary, computationally intensive tasks, such as speech recognition, are offloaded to small data centers, called cloudlets, located close to users [35, 51, 52, 57, 58]. We focus here on a more restricted programmable edge environment [7] but believe the performance interference problems between CDN customers presented in this paper are applicable to more general environments like cloudlets, too.

CDN Quality of Service. Due to CDNs' widespread adoption, their performance has been greatly studied [18, 19, 24, 45, 56]. Researchers have developed numerous techniques for improving CDN performance. At the aggregate level, load balancing [25, 45, 53], optimal server placement [42, 47, 55], and replication [27, 43, 62] have been investigated. On individual servers, prior work studied quality of service through multiple lenses, including control theory [17], resource provisioning [26, 63], model-based approaches [32], real-time kernel extensions [50], and fair queuing [29, 40, 44, 54, 61]. These works also differ in the metrics they sought to improve. Here, we leverage the framework of multi-resource fair queuing [36] to improve fairness in a single CDN server.

Multi-resource Fair Queuing. Multi-resource fairness (MRF), also referred to as dominant-resource fairness, is a generalization of max-min fairness to multiple resource types [36, 37]. Fair queuing schedulers were originally developed to enforce max-min fairness between a set of flows sharing a single resource [29, 54], and numerous algorithms have been proposed [29, 40, 44, 54, 61]. Each trades-off between worst-case bounds on fairness, worst-case request latency, and computational overhead.

Multi-resource fair queuing (MRFQ) [36] first extended one fair queuing algorithm [40], to multiple resources. Since then, more MRFQ schedulers have been developed [36, 68–70]. However, these advancements are largely theoretical; the algorithms either make unrealistic assumptions about the underlying system [36, 69] or as we show, risk under-utilizing parallel system resources, e.g., multi-core CPUs

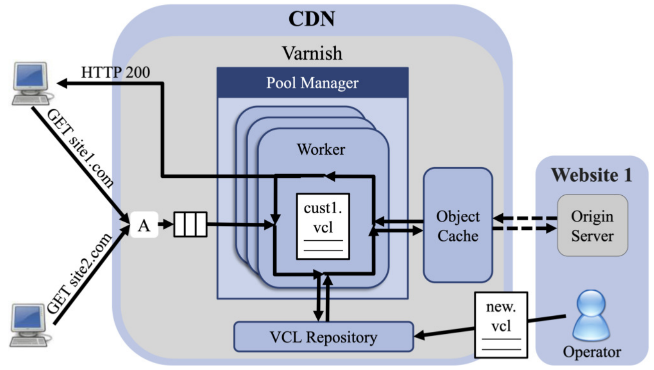


Figure 1: Overview of Varnish's request processing. "A" denotes Varnish's acceptor thread. We highlight the different administrative domains. Dotted lines denote the requests or responses that may not be required for every client request.

[68, 70], making them impractical. One exception is Li and Qian [48], which focused on reducing the overhead of Ghodsi et al. [36] using sketching.

Wang et al. studied the trade-off between multi-resource fairness and resource utilization, both theoretically [67] and in storage systems [66]. However, like other multi-resource scheduling algorithms developed for data centers [23, 28, 38, 65], the latter is not directly applicable to our setting because our system must make online scheduling decisions.

New Kernel APIs. Banga et al. proposed a new system-call interface for decoupling thread and resource monitoring abstractions [21]. However, deploying custom kernel modifications raises issues of portability and maintainability [49]. To maintain Sandpaper's practicality, we thus avoid requiring changes to the underlying operating system.

3 MOTIVATION

In this section, we provide an overview of programmable edge platforms built using Varnish, an open-source HTTP reverse proxy [7]. We then demonstrate that programmability in the CDN's edge enables significant performance interference (to be defined in this section) between customers' requests.

3.1 Programmability at the (CDN's) Edge

Figure 1 shows an overview of a programmable CDN edge environment built with Varnish [7], which is the backbone of several CDNs [1, 2, 14]. Together, a set of Varnish instances service HTTP requests from clients (i.e., end users) around the globe. For instance, Fastly's 40+ locations serve over 7 million requests per second on average [20]. We focus here on a single Varnish instance deployed on a single server, which must handle tens of thousands of requests per second.

In Varnish, a pool of worker threads processes HTTP requests. All requests execute within a single process. One worker accepts new TCP connections; this acceptor thread hands off new requests in FIFO order to the other workers, which process them to completion. For each request, a worker parses its headers and payload, loads and executes the corresponding operator's code (using the request's Host header), assembles the response (from the cache or by contacting origin servers), and transmits the response to the user. Other CDNs use similar architectures [11, 16].

```
1: import geoip2;
2: function VCL_INIT
3:     set countryCodeMap = geoip2.load("/lib/country.mmdb");
4: function VCL_RECV
5:     set req.http.CC = countryCodeMap.lookup(client.ip);
6:     return (hash);
7: function VCL_HASH
8:     append_to_hash_key(req.http.CC);
```

Figure 2: Example VCL to cache multiple location-specific copies of a web page [7]. The syntax deviates slightly from true VCL for clarity of presentation.

Web service operators customize Varnish's request processing by uploading code to Varnish. The code is written in Varnish Configuration Language (VCL). Each operator logically maintains a single VCL. Functions are compiled, linked into Varnish's runtime, and called at specific hooks as the request is processed. Figure 2 shows a VCL that makes Varnish cache location-specific copies of a website as separate objects, allowing users to view location-relevant content. vcl_init and vcl_recv run when the VCL is first loaded and when each request is first received by Varnish, respectively. Line 5 searches a local country code database using the client's IP address. The result of this lookup is stored in a new request header, CC. Line 6 instructs Varnish to next compute its hash function for a cache lookup. The hash function definition on lines 7-8 adds the new CC header to the cache key calculation, in addition to a set of default headers.

3.2 Heterogeneous Resource Demands Cause Performance Interference

Executing code from different customers in a single process can cause performance interference from fine-grained sharing of system resources. Performance interference occurs when a change in one customer's workload, e.g., uploading a new VCL, unfairly affects the performance, either latency or throughput, of another customer's requests. In this section, we demonstrate significant throughput interference in Varnish and show it is caused by heterogeneous resource demands. We later evaluate latency (§6.2).

To start, we construct mock VCLs for four digital publishing services, which are known customers of a major CDN. Each VCL uses the features stated in their customer testimonials [3–6]. A summary of the features employed by each

Customer	Features	CPU Processing	Client-side Network Transmission
C1	Insert relative publication timestamps next to article links.	39.88 (1.96)	22.08 (0.001)
C2	Logs request and response headers to Linux’s system log. Validates JSON web tokens for content personalization.	13.60 (0.88)	81.25 (0.01)
C3	User-Agent header normalization for device-optimized content. Adds Geo-IP information in header for location-specific content.	28.10 (1.19)	92.18 (0.001)
C4	Embed JSON objects with real-time live blog content.	22.81 (0.99)	36.40 (0.001)

Table 1: The features used by 4 mock CDN customers and the resulting CPU and network processing times (in microseconds) for a client request to the customer’s site. Standard deviations for processing times are shown in parentheses. A more detailed explanation of the features and a precise definition of processing time are below.

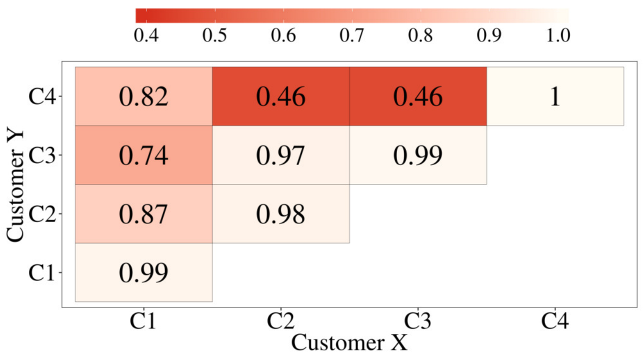


Figure 3: Ratio of the minimum to the maximum of normalized throughput for each pair of customers. A ratio of 1 indicates fair reductions in throughput.

customer is shown in Table 1. C1 uses VCL to insert relative publication timestamps, e.g., “published 5 minutes ago,” next to article links on their page. C2 uses VCL to show users different content based on the user’s privileges. C3 uses VCL to cache location-specific copies of their website as separate objects and optimize the copies for the user’s device. Finally, C4 uses VCL to power their live blog pages, ensuring users always load the latest blog content. We also scrape a copy of each customer’s web page. We henceforth refer to the pair of a VCL and a web page as a *customer*. Note that additional customers can only magnify the potential for performance interference demonstrated here.

To measure performance interference between pairs of customers c_1 and c_2, we use a closed-loop [60] client [39] with 100 threads per customer to measure their *normalized throughput*, $\mu_{c_1,c_2}/\mu_{c_1}$, where μ_{c_1,c_2} is c_1’s mean throughput when c_1 and c_2’s requests compete for system resources in Varnish and μ_{c_1} is c_1’s mean throughput in isolation. In a fair, single-resource system, we expect each customer’s normalized throughput to be 0.5; processing at the resource would be split evenly between the two customers. In a fair multi-resource system, their normalized throughputs are equal but may be greater than 0.5 because heterogeneous resource demands can lead to increases in aggregate throughput.

Far from implementing a fair system, Figure 3 shows that some customers are treated unfairly in Varnish. For each pair, we plot the ratio of the lesser to the greater of the two normalized throughputs. In the most extreme case, C3 vs. C4, the normalized throughputs were 0.68 and 0.31, respectively; C3’s throughput dropped by only 32% while C4’s dropped by 69%. Further, C1’s throughput dropped significantly more than other customers; the normalized throughputs were 0.62 and 0.72; 0.55 and 0.74; and 0.50 and 0.61 for C1 vs. C2, C1 vs. C3, and C1 vs. C4, respectively. §6 contains additional details on our experimental setup and further discussion of these results (§6.1).

We also observed similar performance interference between customers with NGINX, another programmable cache using an event-driven software architecture [15], suggesting these problems are general to a wider range of programmable edge environments. Further, our communications with engineers at Fastly, a leading CDN, confirmed that performance interference is indeed a problem they have observed in production [8]. For instance, engineers at Fastly have observed cases where customer-supplied VCL logic inadvertently triggered pathological performance cases in Fastly’s fork of Varnish, and bugs in vendor-supplied libraries have resulted in inefficient use of synchronization primitives [8].

To investigate the source of the performance interference, we use *processing time* [36] to estimate requests’ demands on system resources. A request’s processing time at a resource is the amount of time, in microseconds, the resource takes to finish servicing it in isolation, divided by the resource’s parallel processing capacity. For instance, a request’s CPU processing time at an 8-core CPU is the time it takes one core to service one request, divided by 8. A request’s largest processing time is its bottleneck resource [36].

We estimate request processing times on two resources: the CPU and the outgoing client-side NIC (one direction of the client-side, full-duplex NIC). Other system resources contributed negligible processing times, so we omit them. To simplify our analysis, we assume multiple resources do not process the same request simultaneously. For the CPU, we calculate $\frac{I}{RC}$, where I is the number of instructions retired while processing the request (measured using Linux's performance monitoring API [72]), R is the CPU's clock rate, and C is the system's total cores. For the NIC, we calculate $\frac{b}{T} + \frac{h}{T}\left\lceil \frac{b}{\text{MTU}-h} \right\rceil$, where b is the number of data bits transmitted (as returned by system calls), h is the size of the TCP/IP header (in bits), and T is the NIC's transmission rate. We assume an MTU of 1500 bytes and a 64-byte TCP/IP header.

Table 1 shows each customer's request processing times. To ensure the assumptions above do not invalidate our analysis, we confirmed that the bottleneck resources predicted by Table 1 match each request's true bottleneck resource. We see that requests from different customers have different bottleneck resources. More importantly, from Figure 3 and Table 1, we see that performance interference occurs between customers with different bottleneck resources, not just between those with the same bottleneck resource.

It may seem that enabling Linux's existing (single-resource) fair schedulers on the CPU and the NIC would help mitigate the performance interference problem [41, 49]. While we admit this is true in cases where all customers' have the same bottleneck resources, such *per-resource fairness* [36] has two disadvantages: First, it does not provide strategy-proofness [36]. This is defined in the next section but the key take-away is that a CDN should not have to trust customers to be well-behaved to enforce fairness. Second, there are resources, such as memory bandwidth, for which it is not obvious how to implement a fair scheduler given the parallel nature of current hardware. This limits the ability to extend this approach to additional resources.

Given the observed performance interference between customers with different bottleneck resources, we argue that the performance interference problem in programmable edge proxies is best studied through the lens of multi-resource fairness (MRF) [37] and solved using techniques from the multi-resource fair queuing (MRFQ) literature.

4 KEY CHALLENGES

In this section, we first provide additional background on MRF [37] and MRFQ [36] and then highlight two challenges that make it difficult to directly apply existing multi-resource schedulers to Varnish. The first challenge arises from operating an MRFQ scheduler at user level, above the operating system's own resource schedulers. The second stems from the mismatch between real system behavior and the theoretical system model assumed by existing MRFQ work.

4.1 Preliminaries

A request's *dominant resource* is the resource requiring the most processing time. For instance, Table 1 shows that the four customer's dominant resources are CPU, network, network, and network, respectively. An MRF scheduler ensures each flow (i.e., stream of requests to a single customer's service) in a set of competing flows receives equal processing time on their respective bottleneck resources. For example, C1 may receive 30% of the CPU's processing time while C2, C3, and C4 each receive 30% of the NIC's.

Enforcing multi-resource fairness between N flows provides three important properties [36, 37]:

(1) *share guarantees*: each flow will receive at least $\frac{1}{N}$ of the system's resources;
(2) *strategy-proofness*: no flow can gain more than their fair share of the system's resources by manipulating their workload (e.g., changing their VCL); and
(3) *work conservation*: when there are queued requests, idle system resources will always be used if doing so increases some flow's throughput.

The first two properties make MRF highly desirable for mitigating the performance interference observed in the previous section. Unfortunately, as we will show, although the third property holds within the theoretical models of the original papers, it fails to hold in practice, resulting in underutilization of system resources.

Multi-resource Round Robin. Round-robin schedulers, like multi-resource round robin (MR^3) [68], implement dequeue operations, which execute once for every request, with $O(1)$ time complexity. Other MRFQ schedulers have $O(\log |Q_{ac}|)$ dequeue operations [36, 69], where Q_{ac} is the set of queued, or active, customers. Because Sandpaper must handle peak loads of up to hundreds of thousands of requests destined for thousands of customers' services, we use MR^3 as the basis for Sandpaper.

Similar to other round robin schedulers, for each customer, MR^3 maintains a FIFO input queue and a bank account to track resource usage [44, 61, 68]. Customer flows receive service in a fixed order over a series of rounds. Newly arriving customers are appended to the end of this order. In each round, a quantum u is deposited into each customer's account. MR^3 releases a customer's requests until the customer's account no longer has enough value to cover the dominant processing time of the request at the head of their queue. MR^3 then moves on to the next customer. Customers may overdraw from their account by up to 1 request in each round, but their excess consumption is deducted in the next round. Unlike other round robin schedulers that use a fixed-size quantum [61], MR^3's quantum size u is set at the start of each round to the maximum excess consumption incurred by any customer in the previous round.

4.2 Application-level Scheduling

Enforcing multi-resource fairness from within Varnish's application runtime has practical advantages: First, it is portable across hardware. Second, it avoids imposing additional burden on other applications running on the same machines as

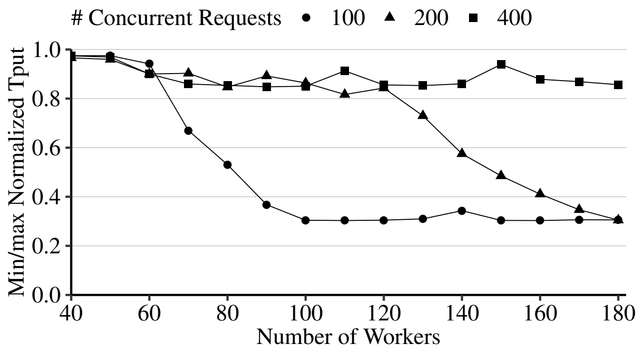


Figure 4: MR^3's effectiveness decreases as the number of workers in Varnish's worker pool approaches the number of concurrent requests being processed.

Varnish. For instance, Fastly deploys multiple applications, such as a DNS resolver, alongside Varnish [22]. But deploying a scheduler at the application-level presents challenges because the new scheduler will run atop the OS's existing resource schedulers.

The existing MRFQ literature assumes that both the scheduler has complete control over the system's resources and the resources process requests under a non-preemptive FIFO discipline [36, 69, 70]. In contrast, an application-level MRFQ scheduler has limited control over the operating system's resource schedulers and thus resources. For instance, Linux's completely fair scheduler (CFS) manages thread (i.e., worker) execution [49]. As a result, an application-level MRFQ scheduler cannot force the CPU to start processing a request; the CFS decides which worker actually receives time on the CPU. Further, the CFS approximates generalized processor sharing (GPS), not a FIFO queue [49, 54]. It divides CPU time equally between runnable threads, and newly runnable threads immediately start receiving processing time. These disparities hinder any application-level MRFQ scheduler's ability to enforce fairness by making it difficult to perform fine-grained detection of overloaded system resources.

Figure 4 demonstrates this problem. We implement MR^3 [68] in Varnish; our experimental setup is otherwise the same as described in §3.2 and §6. We again plot the ratio of the normalized throughputs (so closer to 1 is fairer) for a CPU- and a network-bound customer while varying the maximum size of the worker pool and the maximum number of concurrent requests being processed (i.e., the number of closed-loop [60] clients per customer). With fewer than 40 workers, neither customer's requests saturate their bottleneck resource, so we omit this range. As the number of workers in the pool approaches the number of requests, MR^3 becomes less effective at enforcing fairness between customers. We discuss our key insight and approach to tackling this issue in §5.2.

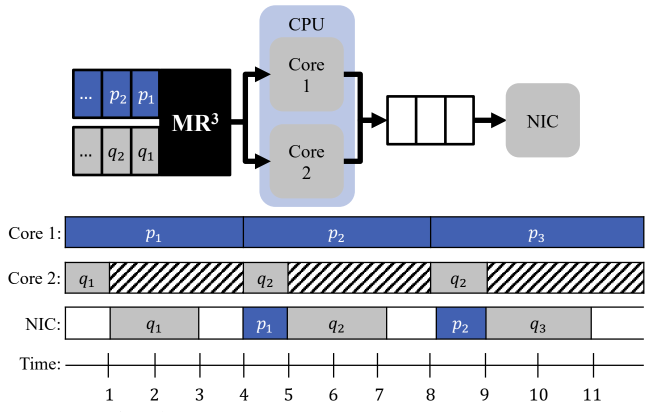


Figure 5: (top) Simplified model of Varnish processing with MR^3. P's requests require 2 and 1 units of CPU and NIC processing time, and Q's require 1 and 2, respectively. (bottom) First 12 microseconds of processing schedule when P and Q share the system above.

4.3 Parallel Resources

We also find that MR^3 (and similar variants) can result in poor utilization of resources because it limits the number of concurrently processed requests [68, 70].

Figure 5 provides the key intuition for this problem using a simplified model of our system with MR^3. We have requests from 2 customers, P and Q, and 2 resources, a CPU and a NIC. The CPU has 2 identical cores. Unlike in our real system, assume MR^3 completely controls the system's resources. P's requests require 2 microseconds of CPU processing and 1 for NIC transmission, and Q's requests require 1 and 2 microseconds, respectively. Recall that since there are 2 cores, the time it takes 1 core to process a request is double the processing time above.

Fig. 5 (bottom) shows the first 12 microseconds of the schedule as MR^3 processes a backlog of 4 requests from each of P and Q. Before releasing the next request from customer c in round R to be processed, MR^3 must wait until at least one of customer c's requests from round $R-1$ starts receiving service on the final resource, the link [68]. Because of this mechanism, one of the cores remains almost entirely idle despite requests being available for processing; these idle cycles are highlighted with a diagonal pattern. This problem is especially pronounced with high degrees of parallelism (e.g., a large multi-core server) and with a small number of customers. Overloaded systems caused by a small number of customers is common for CDNs, for instance during flash crowds, and since server deployment costs directly affect their operating costs, maintaining high resource utilization is critical. Unfortunately, removing this mechanism (and similar blocking mechanisms in other schedulers [36]) causes unbounded worst-case unfairness (i.e., the Relative Fairness

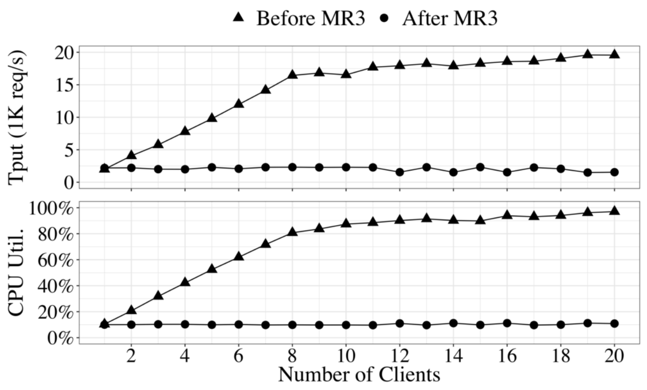


Figure 6: Varnish's throughput (top) and CPU utilization (bottom) cease to scale with the number of concurrent connections after implementing MR^3.

Bound [36, 68, 69] will be infinite) with other workloads. We use a simulator to confirm this for both [68] and [36].

Figure 6 demonstrates the magnitude of this issue in practice. It plots the aggregate throughput (top) and CPU utilization (bottom) before and after implementing MR^3 in Varnish as we vary the number of clients requesting a single customer's website. Varnish is allotted 8 cores. MR^3 results in an enormous reduction in throughput compared to Varnish's original capacity. Further, the low CPU utilization shows that the problem is not scheduler overhead but that Varnish cannot leverage the CPU's parallel cores. We discuss our key insight for tackling this under-utilization problem in §5.3.

5 DESIGN

We begin this section with a brief overview of Sandpaper and then discuss each of Sandpaper's components in turn.

5.1 System Overview

Sandpaper comprises two new components: a worker pool size manager (replacing Varnish's existing manager) and a request releaser (replacing the FIFO request queue). Both are completely contained within the Varnish application runtime and do not require changes to the underlying operating system. Figure 7 highlights the key changes to Varnish's request processing architecture, and Figure 8 presents the details of the algorithm.

Problem Formulation. A request's customer is defined by its Host header. While processing request i to customer c's service, denoted p_c^i, Varnish executes customer c's VCL and loads their web page. The server hosting Varnish comprises an indexed set of resources $R = \{r_1, \ldots, r_m\}$. When p_c^i is processed, its execution is characterized by resource profile $S_c^i = \{s_{c,1}^i, \ldots, s_{c,m}^i\}$, an indexed set of processing times. We do not differentiate between execution explicitly triggered by a customer's VCL and any supporting execution in Varnish's runtime when calculating resource profiles and assume resource profiles are not known a priori. Given a set of customers and their VCL configurations, we wish to enforce MRF between customers.

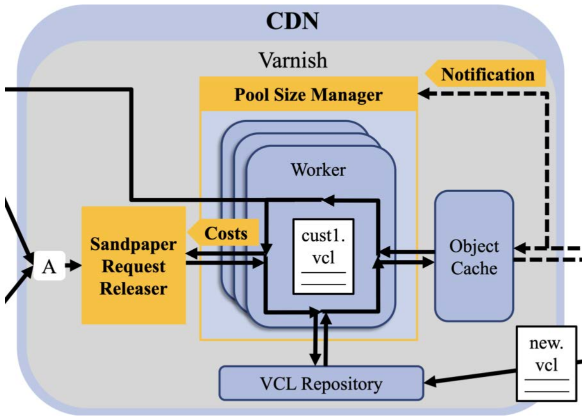


Figure 7: Sandpaper adds two components to Varnish: a request releaser and a worker pool size manager.

Worker Pool Size Manager. This component addresses the issues raised in §4.2. Sandpaper provides a more practical choice in the trade-off between resource utilization and multi-resource fairness. As Figure 4 demonstrated, using a nearly unlimited number of workers, as Varnish does by default, inhibits an application-level MRFQ scheduler's ability to enforce MRF. On the other hand, using too few workers risks not being able to saturate large, multi-core CPUs that are common in modern servers. Thus, Sandpaper's new pool size manager is responsible for regulating the number of workers in the worker pool. It aims to keep the number of active workers as small as possible, to provide the best fairness guarantees, without under-utilizing the server's resources.

Request Releaser. As shown in §4.3, even with ample workers, existing MRFQ schedulers can still under-utilize systems with parallel and pipelined resources because they limit the number of concurrent requests in the system. Thus, we develop Sandpaper's request releaser, which is a new, practical variant of MR^3 [68] that addresses this issue and can saturate large parallel resources while still enforcing fairness between customers. Requests are enqueued in the request releaser when they arrive in the system. The request releaser is then responsible for reordering requests and passing them to idle workers. Upon completion, workers report requests' processing times to the request releaser.

Provable Guarantees. Finally, while striving to improve the practicality of existing MRFQ schedulers, we also want to retain their formal guarantees. We highlight some of the key theoretical results in §5.2 and §5.3 but leave the details of the proofs to the Appendix. The design of Sandpaper's request releaser stems from our key observation that under a more realistic system model, one that includes properties such as threaded execution and blocking network I/O, we can still prove that Sandpaper guarantees fairness despite its relaxations (Appendix A). However, our new analytical results show that Sandpaper's worst-case unfairness is directly proportional to the number of active workers in the worker pool. Thus, we also provide analytical results proving our new pool size manager will not result in an unbounded pool size and thus unbounded unfairness (Appendix B). Although Sandpaper's formal guarantees are much weaker than other schedulers, Sandpaper achieves much better system utilization and performance. Further, we show in our evaluation that fairness in practice is often much better than the worst-case bounds (§6.3).

5.2 Worker Pool Size Manager

Figure 4 suggests that to allow any application-level MRFQ scheduler to enforce fairness: (1) we must limit the size of the worker pool; and (2) the number of workers in the pool must be less than the number of concurrent requests in the system. Since we do not know the number of concurrent requests a priori, we thus must set the worker pool size to be as small as possible to provide the best fairness guarantees. However, a small worker pool risks under-utilizing the system's parallel resources. Empirically, we find with a worker-to-core ratio of 10:1, Varnish can saturate our machine's CPUs, even with the least-demanding VCL and a 1000-byte HTTP object. Thus, Sandpaper simply sets the size of the worker pool $W = 10C$, where C is the number of cores.

But keeping the number of workers lower than the maximum number of concurrent requests poses a potential risk because workers can block on high-latency content fetches from origin servers. Varnish may run out of workers if too many requests block. Further, as mentioned in §5.1, Sandpaper requires blocking network I/O to formally guarantee fairness. Therefore, before a request fetches content from an origin server, it notifies the scheduler, which spawns a new worker for the pool. The new worker begins processing other requests, and upon completion, the original worker terminates, returning the pool to its original size.

We thus need to show that the worker pool's growth under this scheme is bounded because Sandpaper's fairness guarantee is a function of W. Assuming request arrival is a Poisson process with rate λ and content fetch latency is independent and exponentially distributed with mean $\frac{1}{\mu}$, we use a continuous-time Markov chain to derive an analytical result (Appendix B) proving that the expected number of extra workers spawned as a result of content fetch blocking is bounded. Since origin servers process requests in parallel, we derive that the expected number of extra workers in the system is $\frac{\lambda p_m}{\mu}$, where p_m is the probability of a cache miss. Suppose $\lambda = 200{,}000$ req/s, $\frac{1}{\mu} = 0.1$s, and $p_m = 0.15$ [14]; the expected number of additional workers is 3000.

Although unlikely, all of these workers could rejoin the pool at once. However, recall that from the request releaser's perspective, these requests were all supposed to have already received service since they were previously released; their temporary departure from the system simply allowed the requests queued behind them to receive service early. Thus, for simplicity, when the requests rejoin, we let Linux's CFS balance the execution of the already-in-progress requests and of the requests returning from content fetches. This solution does not affect the system's fairness over longer time-scales and avoids any potential starvation.

5.3 Request Releaser

As described in §4.3, MR^3 [68] limits the number of requests released into the system, which can cause under-utilization of parallel resources. We henceforth refer to this mechanism as *end-to-end synchronization.*

End-to-end synchronization is required to prove MR^3's fairness guarantees in the form of a *Relative Fairness Bound* (RFB), which is the difference in processing time received by any two customers on their respective dominant resources during any time interval [36, 68, 70]. Formally, it is:

$$\mathrm{RFB} = \sup_{t_1, t_2 : i, j \in \mathbb{B}(t_1, t_2)} \left| T_i(t_1, t_2) - T_j(t_1, t_2) \right|$$

where $\mathbb{B}(t_1, t_2)$ is the set of active customers in the time interval $[t_1, t_2)$ and $T_i(t_1, t_2)$ is the total dominant processing time received by customer i during the same interval.

Removing end-to-end synchronization would solve our utilization problem but would also invalidate MR^3's guarantees. Our key insight is to leverage system properties to bridge the gap between this seemingly fundamental tension between resource utilization and fairness. Thus, we deploy a variant of MR^3, with end-to-end synchronization removed, and prove its theoretical guarantees in our more realistic system model. We provide the intuition and a proof sketch below.

Four properties of real systems provide a loose form of synchronization while processing requests: First, queued requests are only released when workers are idle. Second, the size of the worker pool limits the number of concurrently processed requests. Third, workers block on network I/O, so they cannot begin processing the next request until their current one finishes NIC transmission. Fourth, the OS's CPU scheduling policy is CFS. When combined, these properties limit the number of requests that can queue between resources and bound the order in which requests complete.

Under a system model that includes these properties, removing MR^3's end-to-end synchronization does not lead to unbounded unfairness. In fact, during backlogged periods where the number of requests is high compared to the number of workers (W), we prove the RFB exists.

```
procedure INIT($C$)                         ▷ Runs once at Varnish start
    SPAWNWORKERS($10C$)

procedure ENQUEUE($p_c^i$)                   ▷ Runs on request arrival
    if $q_c \notin Q_{ac}$ then
        APPEND($Q_{ac}, q_c$)
    APPEND($q_c, p_c^i$)

procedure ONFETCH($p_c^i$)                   ▷ Runs only if fetch required
    SPAWNWORKERS(1)

procedure COMPLETE($w, p_c^i$)               ▷ Runs on request finish
    $s_{c,\uparrow}^i \leftarrow$ MAX($S_c^i$)
    $k_c \leftarrow \alpha s_{c,\uparrow}^i + (1 - \alpha)k_c$
    if REQUIREDFETCH($p_c^i$) then
        DESTROYWORKER($w$)

procedure DEQUEUE
    while true do
        if $R = 0$ then
            $R \leftarrow |Q_{ac}|$
            $u \leftarrow s_{\max}$
            $s_{\max} \leftarrow 0$
        $n \leftarrow$ REMOVEHEAD($Q_{ac}$)
        $s_n \leftarrow s_n - (u + 1)$
        while $s_n < 0$ and $|q_n| > 0$ do
            $w \leftarrow$ WAITUNTIL($WorkerAvailable$)
            $p_n^i \leftarrow$ REMOVEHEAD($q_n$)
            RELEASEREQUEST($w, p_n^i$)
            $s_n \leftarrow s_n + k_n$
        if $|q_n| > 0$ then
            APPEND($Q_{ac}, q_n$)
        else
            $s_n \leftarrow 0$
        $s_{\max} \leftarrow$ MAX($s_{\max}, s_n$)
        $R \leftarrow R - 1$
```

Figure 8: Sandpaper algorithm. Q_{ac} is the set of active customers. R tracks the number of customers left in this round. u is the quantum. n is the index of the next customer to receive service. w is an idle worker to process the request p_n^i. s_n tracks the customer's excess consumption, and k_n is the customer's processing time estimate.

THEOREM 5.1. *Consider any given pair of customers i and j during a backlogged period. We have*

$$|T_i(t_1, t_2) - T_j(t_1, t_2)| \leq 2WL + 2L$$

where L is the maximum processing time across all resources and all customers.

The proof of Theorem 5.1, for simplicity of analysis, makes extra assumptions beyond the four system properties above. Due to space constraints, we defer the details to Appendix A.

6 IMPLEMENTATION & EVALUATION

We use both a testbed and a simulator to evaluate our approach. We start by demonstrating that Sandpaper enforces fairness between customers (§6.1) and improves tail latency (§6.2). Next, we evaluate the request releaser and its design choices (§6.3). Then, we analyze Sandpaper's scalability and performance overhead (§6.4). Finally, we present case studies showing Sandpaper mitigates performance interference in realistic scenarios (§6.5).

Experimental Setup. Our testbed runs on Wisconsin's CloudLab cluster [33]. We use up to 6 machines: up to 4 for load generation, 1 for Varnish [7], and 1 for Apache HTTP server [12]. Each machine has two Intel E5-2660 10-core, 2.60 GHz CPUs and 128 GB DDR4 RAM with hyper-threading disabled. They use dual-port Intel X520 10 Gb NICs. We use wrk [39] for closed-loop [60] load generation. Unless stated otherwise, we restrict Varnish to use 8 physical cores.

Implementation Details. We implement Sandpaper as a modification to Varnish 6.0 (commit `52e8fb1`) [7] in 1026 lines of C code, including whitespace. Our implementation leverages an existing hashmap implementation [34]. We use SimPy [59], a discrete-event simulation framework, to build a simulator of Varnish's request processing in 2687 lines of Python code, including whitespace. We use the same methodology as described in §3.2, including the same performance counters, to estimate request processing times on the CPU and NIC. The request releaser uses an exponentially-weighted moving average with $\alpha = 0.9$ to estimate the current request processing time for each customer. To reduce performance overhead, we sample performance counters for 20% of the requests.

6.1 Fairness & Throughput

We re-run the experiment presented in Figure 3 (§3.2) after implementing Sandpaper in Varnish. We start by measuring each customer's mean throughput in isolation using 100 client threads. Customers' baseline throughputs vary significantly: C1's is 19,568 req/s, C2's is 11,005 req/s, C3's is 10,668 req/s, and C4's is 26,966 req/s. For each pair of customers, c_1 and c_2, we then use 100 clients per customer (200 total) to measure each customer's mean throughput when the pair's requests compete for processing time and calculate each customer's normalized throughput, $\mu_{c_1,c_2}/\mu_{c_1}$.

Figure 9 again plots the min/max ratio of normalized throughputs for each pair of customers. The ratio for nearly all pairs has moved closer to 1, indicating that Varnish with Sandpaper divides its processing time and resources more fairly. We suspect the cases that remained significantly less than 1 are due to Sandpaper's relaxed fairness bounds and the inaccuracy of the processing time estimation. However,

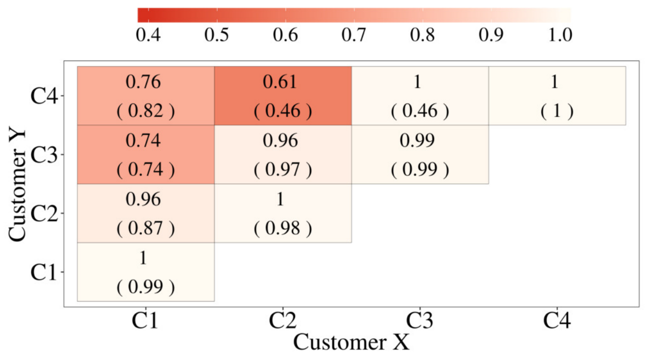


Figure 9: With Sandpaper, customers' are treated more fairly. Nearly all ratios moved towards 1. Original values shown in parentheses.

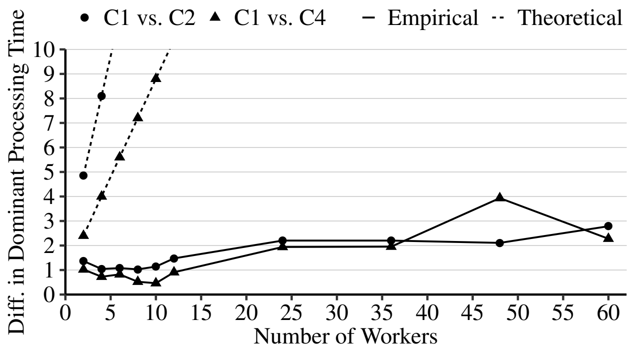


Figure 10: Theoretical and empirical fairness bounds of Sandpaper's request releaser. In practice, Sandpaper provides significantly tighter bounds than predicted by the derived RFB.

even in these cases, Sandpaper has a significant effect on tail latencies, as discussed in the next section. We also verify Sandpaper does not improve fairness by simply slowing down the customer that originally received more processing time. Instead, Sandpaper enables Varnish to better split its processing time among customers.

6.2 Latency

Even in cases where fairness is not enforced perfectly, Sandpaper has a significant effect on tail latency. For instance, in Figure 9, when C1 and C3 compete, C1's 99th percentile latency increases from 39.42 ms to 101.85ms in exchange for a huge reduction in C3's 99th percentile latency, from 4.83 seconds to 19.89 ms. In fact, across all pairs in Figure 9, Sandpaper reduces the maximum 99th percentile latency from 5.77 s, for C2 against C1, to 411.80 ms, for the same.

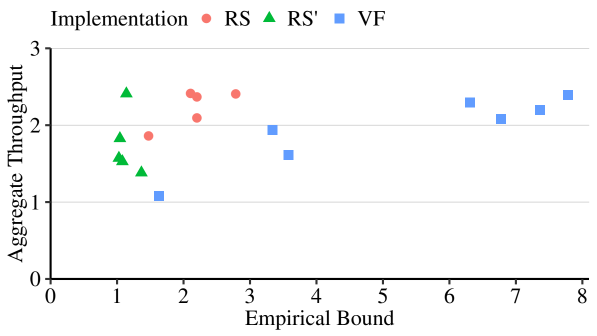
6.3 Evaluating the Request Releaser

We use a simulator to analyze the impact of removing MR^3's end-to-end synchronization on observed fairness. We then compare the performance of this approach against an alternative solution.

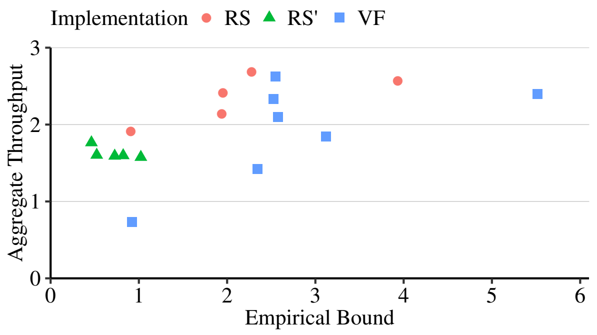
Empirical Fairness Bounds. We simulate a system similar to the one depicted in Figure 5. We set the CPU core count to 12 in order to analyze cases where MR^3 causes under-utilization with our experimental workloads. The primary difference between Figure 5 and our simulator is that I/O calls from threads to the NIC are blocking, more similar to our real system, rather than being placed in a queue before the NIC. We implement generalized processor sharing [54], which Linux's CFS approximates, for our simulated CPU scheduler. Threads are assigned uniformly to each core and share the same scheduling priority. We define the *empirical bound* as the maximum observed difference in dominant resource usage (i.e., the aggregate processing time on the dominant resource) between any two customers.

Since the RFB only describes worst-case unfairness, we investigate the empirical bound of Sandpaper's request releaser in Figure 10. For each experiment, requests from two customers arrive at the scheduler. For each customer, we use the processing times as measured experimentally in Varnish and shown in Table 1. The solid lines show the empirical bound between the two customers as we vary the number of workers (threads) in the worker pool, W. For each pair of simulated customers, we increase the value of W until the aggregate throughput stops increasing. The dotted curves show the derived RFBs (see Appendix A for more details) for the corresponding pairs. Like the RFB, the empirical bound also experiences an upward trend as W increases. However, the empirical bound is significantly smaller than the derived RFB. We note that the pessimistic RFB is only intended to prove Sandpaper's request releaser does not lead to unbounded unfairness; tightness of this bound is not our primary concern.

Alternative Approach. By removing end-to-end synchronization, we obtained the variant of MR^3 that we used in Sandpaper. In this section, we refer to this variant as *remove synchronization* (RS). Despite the existence of a Relative Fairness Bound for RS, it remains difficult to understand or model other performance metrics, such as RS's mean relative fairness or aggregate throughput. Thus, we resort to empirical evaluation and compare it against an alternative variant of MR^3 that introduces *virtual flows* (VF) to improve resource utilization. Specifically, VF splits requests from an active customer i into a group of α virtual customers $i^{(1)}$, $i^{(2)}$, ..., $i^{(\alpha)}$. With a set of Q_{ac} active customers, VF artificially inflates the total number of virtual flows to $\alpha\,|Q_{ac}|$. VF then enforces fairness among customers in each of the α virtual groups (e.g., progress of $i^{(2)}$ and $j^{(2)}$ is governed by the same RFB). Using this approach, we can directly adapt the proof of MR^3's Relative Fairness Bound [68] to show that VF's Relative Fairness Bound also exists.



(a) C1 vs. C2



(b) C1 vs. C4

Figure 11: Comparing aggregate throughput of RS and VF as a function of the observed empirical bound. For a fixed empirical bound, RS generally achieves higher aggregate throughput than VF.

We compare RS and VF in terms of aggregate throughput (i.e., the number of requests processed from all customers per unit time) as a function of the empirical bound. Figure 11 shows the results for two representative pairs of customers. We increase W and α until the aggregate throughput stops increasing. The RS' data points correspond to configurations where W is lower than the core count. The key observation is that RS achieves higher throughput than VF when their empirical unfairness bounds are equal. Second, setting W to a value below the core count results in low aggregate throughput, as expected, since the workers cannot saturate the system's simulated parallel resources. We observe similar patterns with the other pairs of customers.

From this analysis, we conclude that compared to VF, RS achieves higher aggregate throughput when flows reach the same level of empirical dominant resource fairness. Moreover, RS has practical advantages over VF. RS does not underutilize the CPU as long as W is initialized to a sufficiently high value, which Sandpaper's worker pool size manager guarantees. This choice is independent of the number of active customers. For VF, however, the system might remain under-utilized when α is too small and the number of active customers drops. Therefore, we argue that RS is the better and simpler solution for Sandpaper.

6.4 Scalability & Overhead

We investigate Sandpaper's scalability and quantify its performance overhead.

Scalability. To show how well Sandpaper scales using realistic customer VCLs, we compare the maximum throughput (in requests per second) of Varnish before and after introducing Sandpaper. We compare Varnish's aggregate throughput as we increase the number of customers. For a given number of customers N, we generate load using $10N$ connections. Each customer has up to 10 requests being processed by Varnish concurrently. We use C1 as a representative customer but find the results are similar for the others.

Sandpaper scales easily to a large number of customers. Its overhead, the percentage decrease in throughput compared to unmodified Varnish for the same workload, ranges from about 5% with only a handful of customers up to 8.6% with 2000 customers supplying 20,000 concurrent requests.

Overhead. To quantify Sandpaper's overhead, we measure Varnish's throughput on 8 cores with 100 connections using the default (no-op) VCL and a 1KB HTML object. Such requests require very little processing and thus amplify the effects of any overhead introduced by Sandpaper. We find that Linux's perf event subsystem [72] is the largest source of overhead in Sandpaper, and other research confirms the numbers we find here [71]. Simply enabling per-thread performance counters and reading them on every request to estimate each request's processing time reduces this lightweight customer's throughput from 248,767 to 195,190 req/s, a 20% reduction. The overhead comprises two parts: a fixed additional amount of CPU processing per context switch and a variable component, lock overhead, that is proportional the number of counter reads. We reduce this latter piece by sampling 20% of requests, reducing the overhead for this customer to 17%. Other work suggests the remaining overhead can be eliminated [30, 71], but we leave integration of their techniques into Sandpaper for future work.

Because the customer used in these experiments was very lightweight, 17% overhead represents the worst case. Further, the remaining overhead does not change as customers increase the complexity of their VCLs or the size of their HTML objects. For the pairs of customers shown in Figure 9, aggregate throughput ranges from 10,693 req/s (C3 vs. C3) to 27,078 req/s (C4 vs. C4) with Sandpaper, compared to 10,691 req/s to 27,047 req/s originally. In fact, Sandpaper imposes less than 9% overhead on all of our realistic customer workloads.

6.5 Case Studies

We present several case studies to show that Sandpaper mitigates performance isolation in realistic scenarios.

Customer Content Changes. Sandpaper protects customers' performance when other customers make content

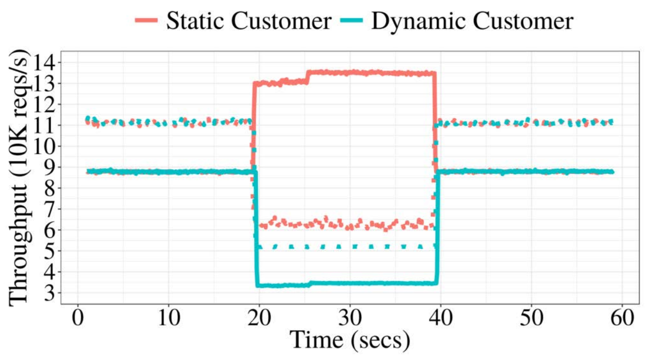


Figure 12: Customer throughput before (dotted) and after (solid) Sandpaper. Originally, the dynamic customer's changes cause a drastic decrease in both customers' throughputs. With Sandpaper, the static customer's throughput is isolated from the changes.

changes. In this experiment, we measure the mean throughput for two synthetic customers, static and dynamic, for 60 seconds. For each customer, we use a closed-loop [60] request generator [39] with 100 client threads. Initially, both the static and the dynamic customer's requests execute identical VCLs and load identical web pages. The CPU originally is the bottleneck resource for both. After 20 seconds, however, the dynamic customer uploads a much larger web page, making the network their bottleneck resource. Finally, the dynamic customer reverts their change after an additional 20 seconds.

In Figure 12, we see that before adding Sandpaper (dotted lines), Varnish fails to mitigate the dynamic customer's performance interference. Because of their identical configurations, each customer's throughput is initially equal, as expected. At time 20, since the dynamic customer's requests now require transmitting more bytes, we expect their throughput to drop. Ideally, however, the static customer, who has not changed their VCL or content, should receive at least the throughput they were receiving prior to the dynamic customer's changes; unfortunately, both customers' throughputs drop precipitously.

With Sandpaper (solid lines), the static customer's throughput is isolated from the dynamic customer's changes. Even better, their throughput now increases when the dynamic customer makes their change because the dynamic customer's bottleneck resource has shifted to the NIC, leaving more processing time available for the static customer's requests on the CPU. The difference in throughput before 20 and after 40 is due to Sandpaper's performance overhead, as discussed in the previous section.

Flash Crowds. Sandpaper also mitigates the impact of flash crowds. We use a similar experimental setup as described above but change the behavior of the dynamic customer. In Figure 13, the dynamic customer experiences three

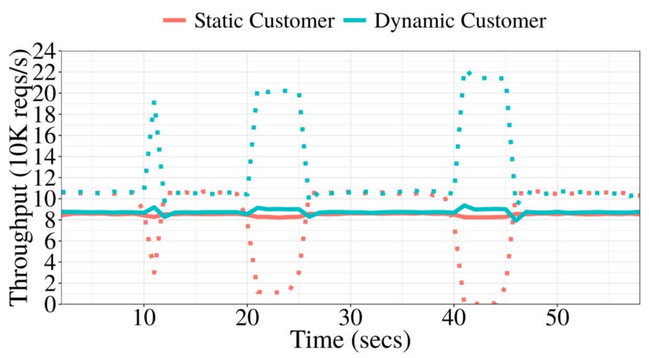


Figure 13: With Sandpaper (solid), customers are more resilient against flash crowds compared to mainline Varnish (dotted).

spikes in the number of connections while the static one sees no change in demand. The first spike occurs at time 10, lasts for 1s, and doubles the dynamic customer's traffic. Comparing the dotted curves (mainline Varnish) and solid curves (Sandpaper), we note that the spike has negligible effect on Sandpaper. The second spike at 20 lasts for 5s and again doubles the traffic. The impact on throughput is now more substantial for mainline Varnish and noticeable in Sandpaper. At time 40, the third spike starts with the same duration as the second but boosts the magnitude to 3X. Its impact is almost identical to the second spike for Sandpaper. Thus, customer performance under Sandpaper is more resilient against flash crowds compared to mainline Varnish.

7 LIMITATIONS

We highlight a few key limitations of our approach: First, not all bottlenecks are caused by contention on shared resources, e.g., different customers' throughputs may be limited by available bandwidth at the origin servers. However, Sandpaper's primary concern is to enforce fairness on the local server's resources; external bottlenecks are beyond the scope of our current approach. Applying Sandpaper in these cases won't produce unfair results but may not produce the equal reduction of throughput competing customers expect. Second, some of Sandpaper's approach and assumptions may not be applicable to systems using event-driven software architectures [13, 15]. However, we expect they are to other CDN edge environments using thread-based architectures [7, 11, 16]. Further, we demonstrated the performance interference problem is general to these other architectures (§3.2) and believe the intuitions from Sandpaper may serve as the basis for a potential solution in event-driven architectures (e.g., by queuing requests before release into an event processor). Finally, Sandpaper's request releaser requires an accurate method for estimating request processing times. For our prototype, we use an exponentially-weighted moving average, but this may not work in all cases. Since the best approach for estimation will be application-specific, we believe development of better estimators is outside our scope.

8 CONCLUSION

We presented Sandpaper, a new multi-resource fair queuing scheduler for enforcing performance isolation between customers in CDN edge platforms. Sandpaper addresses practical deployment challenges in its design; its implementation is entirely contained in user space and does not require modifications to the underlying operating system. Sandpaper also offers a more practical trade-off between multi-resource fairness and resource utilization than existing schedulers. Sandpaper's design is enabled by key insights regarding the differences between theoretical system models and real systems, allowing us to improve the performance of existing schedulers while still providing theoretical guarantees. We demonstrated that Sandpaper performs well in realistic scenarios in mitigating performance interference between workloads based on real customers.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Kurtis Heimerl, for their helpful comments and feedback. This work was supported in part by the NSF under award CNS-1700521 and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] 2012. Varnish Case Study: SFR Telecommunications. (2012). https://www.varnish-software.com/case-studies/

[2] 2015. Varnish Case Study: CacheFly. (2015). https://www.varnish-software.com/case-studies/

[3] 2016. Fastly Case Study: Business Insider. (2016). https://www.fastly.com/industries/

[4] 2016. Fastly Case Study: Fast Company. (2016). https://www.fastly.com/industries/

[5] 2016. Fastly Case Study: The Guardian. (2016). https://www.fastly.com/industries/

[6] 2016. Fastly Case Study: Wired. (2016). https://www.fastly.com/industries/

[7] 2018. Varnish HTTP Cache. (2018). https://varnish-cache.org/

[8] 2019. Personal communication with Fastly engineers.. (2019).

[9] 2019. Akamai Cloudlet Applications. (2019). https://www.akamai.com/us/en/products/performance/cloudlets/

[10] 2019. Akamai Edge Side Includes. (2019). https://www.akamai.com/us/en/support/esi.jsp

[11] 2019. Amazon Web Services Lambda@Edge. (2019). https://aws.amazon.com/lambda/edge/

[12] 2019. Apache HTTP Server. (2019). https://httpd.apache.org/

[13] 2019. Envoy Proxy. (2019). https://www.envoyproxy.io/

[14] 2019. Fastly: Edge Cloud Platform. (2019). https://www.fastly.com/

[15] 2019. NGINX Reverse Proxy. (2019). https://www.nginx.com

[16] 2019. Serverless Computing with Cloudflare Workers. (2019). https://www.cloudflare.com/products/cloudflare-workers/

[17] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. 2002. Performance Guarantees for Web Server End-systems: A Control-theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems* 13, 1 (Jan. 2002).

[18] Vijay K Adhikari, Yang Guo, Fang Hao, Volker Hilt, Zhi-Li Zhang, Matteo Varvello, and Moritz Steiner. 2014. Measurement Study of Netflix, Hulu, and a Tale of Three CDNs. *IEEE/ACM Transactions on Networking (TON)* 23, 6 (2014).

[19] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. 2012. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*.

[20] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[21] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[22] Artur Bergman. 2017. What is Fastly? (2017). https://www.slideshare.net/Fastly/fastly-ceo-artur-bergman-at-altitude-nyc

[23] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Hierarchical Scheduling for Diverse Datacenter Workloads. In *Proc. ACM Symposium on Cloud Computing (SOCC)*.

[24] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*.

[25] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. 2002. The State of the Art in Locally Distributed Web-server Systems. *ACM Computing Surveys (CSUR)* 34, 2 (June 2002).

[26] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. 2001. Managing Energy and Server Resources in Hosting Centers. *ACM SIGOPS Operating Systems Review* 35, 5 (Oct. 2001).

[27] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. 2002. Dynamic Replica Placement for Scalable Content Delivery. In *Proc. Peer-to-Peer Systems*.

[28] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[29] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *ACM Computer Communications Review (CCR)* 19, 4 (1989).

[30] John Demme and Simha Sethumadhavan. 2011. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In *Proc. International Symposium on Computer Architecture (ISCA)*.

[31] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. 2002. Globally Distributed Content Delivery. *IEEE Internet Computing* 6, 5 (2002).

[32] Ronald P Doyle, Jeffrey S Chase, Omer M Asad, Wei Jin, and Amin Vahdat. 2003. Model-Based Resource Provisioning in a Web Service Utility.. In *Proc. USENIX Symposium on Internet Technologies and Systems*.

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proc. USENIX Annual Technical Conference (ATC)*.

[34] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[35] Jason Flinn and M. Satyanarayanan. 1999. Energy-aware Adaptation for Mobile Applications. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*.

[36] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-Resource Fair Queueing for Packet Processing. *ACM Computer Communications Review (CCR)* 42, 4 (2012).

[37] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[38] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. In *Proc. ACM European Conference on Computer Systems (EuroSys)*.

[39] Will Glozer. 2018. Wrk: Modern HTTP Benchmarking Tool. (2018). https://github.com/wg/wrk

[40] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proc. ACM SIGCOMM Conference*.

[41] Bert Hubert. 2001. Linux man pages: tc. (2001). http://man7.org/linux/man-pages/man8/tc.8.html

[42] Kamal Jain, Mohammad Mahdian, and Amin Saberi. 2002. A New Greedy Approach for Facility Location Problems. In *Proc. ACM Symposium on Theory of Computing (STOC) (STOC ’02)*.

[43] Jussi Kangasharju, James Roberts, and Keith W Ross. 2002. Object Replication Strategies in Content Distribution Networks. *Computer Communications* 25, 4 (2002).

[44] S. S. Kanhere, H. Sethu, and A. B. Parekh. 2002. Fair and Efficient Packet Scheduling Using Elastic Round Robin. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 13, 3 (2002).

[45] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. 2001. On the Use and Performance of Content Distribution Networks. In *Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*.

[46] I. Lazar and W. Terrill. 2001. Exploring Content Delivery Networking. *IT Professional* 3, 4 (July 2001).

[47] Bo Li, M. J. Golin, G. F. Italiano, Xin Deng, and K. Sohraby. 1999. On the Optimal Placement of Web Proxies in the Internet. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, Vol. 3.

[48] X. Li and C. Qian. 2015. Low-Complexity Multi-Resource Packet Scheduling for Network Function Virtualization. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*.

[49] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proc. ACM European Conference on Computer Systems (EuroSys)*.

[50] Klara Nahrstedt, Hao-hua Chu, and Srinivas Narayan. 1998. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High Speed Networks* 7, 3-4 (Dec. 1998).

[51] P. Nguyen, T. Elgamal, S. Konstanty, T. Nicholson, S. Turner, P. Su, K. Nahrstedt, T. Spila, R. H. Campbell, J. Dallesasse, M. Chan, and K. McHenry. 2019. Bracelet: Edge-Cloud Microservice Infrastructure for Aging Scientific Instruments. In *Proc. International Conference on Computing, Networking and Communications (ICNC)*.

[52] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, Eric Tilton, Jason Flinn, and Kevin Walker. 1997. Agile Application-Aware Adaptation for Mobility. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*.

[53] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. 2010. The Akamai Network: A Platform for High-performance Internet Applications. *ACM SIGOPS Operating Systems Review* 44, 3 (Aug. 2010).

[54] A. K. Parekh and R. G. Gallager. 1993. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking (TON)* 1, 3 (1993).

[55] Lili Qiu, Venkata N Padmanabhan, and Geoffrey M Voelker. 2001. On the Placement of Web Server Replicas. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*, Vol. 3.

[56] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. 2002. An Analysis of Internet Content Delivery Systems. *ACM SIGOPS Operating Systems Review* 36, SI (Dec. 2002).

[57] Mahadev Satyanarayanan. 2001. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications* 8, 4 (2001).

[58] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017).

[59] Stefan Scherfke and Ontje Lünsdorf. 2018. SimPy: Discrete Event Simulation for Python. (2018). https://simpy.readthedocs.io/

[60] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[61] M. Shreedhar and G. Varghese. 1996. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking (TON)* 4, 3 (1996).

[62] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. 2004. Replication for Web Hosting Systems. *ACM Comput. Surv.* 36, 3 (Sept. 2004).

[63] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3, 1 (March 2008).

[64] A. Vakali and G. Pallis. 2003. Content Delivery Networks: Status and Trends. *IEEE Internet Computing* 7, 6 (Nov. 2003).

[65] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proc. ACM Symposium on Cloud Computing (SOCC)*.

[66] Hui Wang and Peter Varman. 2014. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*.

[67] Wei Wang, Chen Feng, Baochun Li, and Ben Liang. 2014. On the Fairness-Efficiency Tradeoff for Packet Processing with Multiple Resources. In *Proc. ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[68] W. Wang, B. Li, and B. Liang. 2013. Multi-Resource Round Robin: A Low Complexity Packet Scheduler with Dominant Resource Fairness. In *Proc. IEEE International Conference on Network Protocols (ICNP)*.

[69] W. Wang, B. Liang, and B. Li. 2013. Multi-Resource Generalized Processor Sharing for Packet Processing. In *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*.

[70] W. Wang, B. Liang, and B. Li. 2014. Low Complexity Multi-Resource Fair Queueing with Bounded Delay. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*.

[71] Vincent M. Weaver. 2015. Self-Monitoring Overhead of the Linux perf_event Performance Counter Interface. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[72] Vincent M. Weaver. 2018. Linux man pages: perf_event_open. (2018). http://man7.org/linux/man-pages/man2/perf_event_open.2.html

A RELATIVE FAIRNESS BOUND OF RS

In this section, we derive analytical bounds on the fairness of RS. We start by explicitly presenting the list of assumptions about the analytical model in Figure 5.

(1) The CPU is multi-core and employs the CFS scheduler.
(2) The CPU has W threads to support parallel processing of requests in Varnish. Each thread can handle one request at a time and shares the same weight under CFS. We further assume the NIC can only process one request at a time.
(3) No buffer exits between the scheduler and the CPU, and between the CPU and the NIC. The scheduler releases a request iff a core is idle. After a thread completes processing a request, it remains blocked until the NIC is ready to handle the request. The core continues execution on other runnable threads.
(4) For each active flow, its packets share the same resource profile during any of its backlogged period. Note that it is slightly stronger than assuming that all flows are dominant-resource monotonic (the flow's dominant resource does not change during any of its backlogged periods) in [37, 68].
(5) There are more than W requests to be processed in a backlog period.

We start with the notations used in the analysis.

(1) L_i be the maximum request processing time of flow i across all resources. $L = \max_i\{L_i\}$.
(2) D_i^k denotes the dominant service flow i receives in round k.
(3) ϕ_i^k denotes the number of requests released from flow i in round k.
(4) P_i^k denotes the sequence of requests released from flow i in round k.

We assume the following holds under CFS scheduling:

Lemma A.1. *For requests with the same CPU processing cost, the order by which they begin processing at the CPU is the same as the order by which they complete processing at the CPU. It also implies that requests from the same flow are processed in FIFO order at both CPU and NIC.*

Using the ordering property in Lemma A.1, we can examine the progress gap between resources by only looking at the highest round number seen for each flow. At a given time t, we measure the progress of Flow i on resource r through a function $h_r(i,t)$, that returns the highest round number of all packets from i seen by r. We define the progress gap of flows on r as

$$\max_i h_r(i,t) - \min_i h_r(i,t)$$

Intuitively, it is the difference between the highest round number of the fastest flow and that of the slowest flow on r. Similarly, we define the progress gap between two resources, r and r', as

$$\max\left(\max_i h_r(i,t) - \min_i h_{r'}(i,t), \max_i h_{r'}(i,t) - \min_i h_r(i,t)\right)$$

Following the notations in Figure 8, we denote the excess consumption of flow i in round k as s_i^k and the maximum excess in round k as s_{max}^k. We note that the following lemmas and corollaries are proven in [68] and they still hold in RS as they do not depend on end-to-end synchronization.

Corollary A.2. *In each round, at least one request from every flow is released by the scheduler.*

Lemma A.3. *$s_i^k \leq L_i$ for any given flow i and round k.*

Lemma A.4. *$s_{max}^k \leq L$ for any given round k.*

Lemma A.5. *$D_i^k = s_{max}^{k-1} - s_i^{k-1} + s_i^k$ for any given flow i and round k. Note that $s_i^0 = 0$ and $s_{max}^0 = 0$.*

Corollary A.6. *$D_i^k \leq 2L$ for any given flow i and round k.*

Before we derive the RFB for RS, we show that the following holds.

Lemma A.7. *Consider any given pair of flows i and j during a backlogged period. Let μ and ν be the smallest and largest round number for all requests from i in the system, respectively. Then all requests from P_j^k where $k < \mu - W$ have exited and all requests from P_j^k where $k > \nu + W$ have not been scheduled.*

(1) All requests from P_j^k where $k < \mu - W$ have exited.
(2) All requests from P_j^k where $k > \nu + W$ have not been scheduled.

Proof. We prove by contradiction.

For Statement 1, suppose that a request from P_j^k where $k < \mu - W$ is still in the system. Since the system has already proceeded beyond round μ, requests from P_j^k, P_j^{k+1}, $\cdots$, $P_j^{\mu-W-1}$ have been released. By Lemma A.1 and Corollary A.2, more than W requests from flow j is being served in the system. In addition, at least one request resides in the system from $P_i\mu$. It follows that the system contains over $W+1$ requests, which is infeasible given that the CPU maintains up to W threads and the NIC processes one request at a time.

For Statement 2, suppose that a request from P_j^k where $k > \nu + W$ is still in the system. Since the system has reached round $\nu + W$, requests from $P_i^{\nu+1}$, $P_i^{\nu+2}$, $\cdots$, $P_i^{\mu+W-1}$ have been released. By Lemma A.1 and Corollary A.2, more than W requests from flow i is being served in the system. In addition, at least one request resides in the system from $P_i\nu$. It follows that the system contains over $W+1$ requests, which again violates the capacity constraint. □

Now we prove the RFB for RS. Let $T_i(t_1, t_2)$ be the dominant resource service flow i receives from time t_1 till t_2. We show that the following theorem holds. The technique is adapted from [68].

Theorem A.8. *Consider any given pair of flows A and B during a backlogged period. We have*

$$|T_B(t_1,t_2) - T_A(t_1,t_2)| \leq 2WL + L_A + L_B + 2\max(\omega(A,B), \omega(B,A))$$

where $\omega(i,j)$ denote the maximum amount of dominant resource service j receives during the period when i receives within one round.

Sketch of Proof. Suppose that from time t_1 to t_2, A receives dominant resource service from round b to e. We use τ_1 and τ_2 to denote the time instant when the first request(s) in P_A^b starts consuming dominant resource and when the last request(s) in P_A^e finishes consuming.

By Lemma A.1, all requests from P_A^b to P_B^e are processed in this period and no other requests from A are processed between τ_1 and τ_2. Note that it applies all resources, including the CPU where cores might be processing these requests across different rounds in parallel. We have

$$T_A(\tau_1,\tau_2) = \sum_{k=b}^{e} D_A^k = \sum_{k=b}^{e} s_{max}^{k-1} - s_A^{b-1} + s_A^f$$

.

To compute $T_B(\tau_1, \tau_2)$ and $|T_B(\tau_1, \tau_2) - T_A(\tau_1, \tau_2)|$, we consider two cases.

Case 1: $b - e \leq 2W$**.** Let $\delta \geq 0$ be the lower bound of $T_B(\tau_1, \tau_2)$ such that $T_B(\tau_1, \tau_2) \geq \delta$.
Now consider the upper bound of $T_B(\tau_1, \tau_2)$. When the first request in P_A^b starts processing on the dominant resource, by Lemma A.7 requests in P_B^k where $k < b - W$ have already exited. When the last request in P_A^e finishes processing, requests in P_B^k where $k > e + W$ are not scheduled. Thus, the amount of dominant resource received by B is upper bounded by

$$T_B(\tau_1,\tau_2) \leq \sum_{k=b-W}^{e+W} D_B^k = \sum_{k=b-W}^{e+W} s_{max}^{k-1} - s_A^{b-W-1} + s_A^{e+W}$$

Combining the upper and lower bounds of $T_B(\tau_1, \tau_2)$ yields

$$\begin{aligned}
&T_B(\tau_1,\tau_2) - T_A(\tau_1,\tau_2)\\
&\leq \sum_{k=b-W}^{e+W} D_B^k - \sum_{k=b}^{e} D_A^k\\
&= \sum_{k=b-W}^{e+W} s_{max}^{k-1} - s_A^{b-W-1} + s_A^{e+W} - (\sum_{k=b}^{e} s_{max}^{k-1} - s_A^{b-1} + s_A^e)\\
&\leq \sum_{k=b-W}^{b-1} s_{max}^{k-1} + \sum_{k=e+1}^{e+W} s_{max}^{k-1} - s_A^{b-W-1} + s_B^{e+W} + s_A^{b-1} - s_A^e\\
&\leq WL + WL - 0 + L_B + L_A - 0\\
&= 2WL + L_B + L_A
\end{aligned}$$

and

$$\begin{aligned}
&T_B(\tau_1,\tau_2) - T_A(\tau_1,\tau_2)\\
&\geq \delta - \sum_{k=b}^{e} D_A^k\\
&\geq 0 - (\sum_{k=b}^{e} s_{max}^{k-1} - s_A^{b-1} + s_A^e)\\
&\geq -(e-b)\cdot L + 0 - L_A\\
&\geq -2WL - L_A
\end{aligned}$$

It follows that $|T_B(\tau_1, \tau_2) - T_A(\tau_1, \tau_2)|$ is bounded by $2WL + L_B + L_A$ when $b - e \leq 2W$.

Case 2: $b - e > 2W$ In this case, we note that the upper bound remains the same for $T_B(\tau_1, \tau_2)$. As for the lower bound, Lemma A.7 requests in $P_B^{b+W}, \cdots, P_B^{e-W}$ receives dominant resource service during (τ_1, τ_2). It follows that

$$\begin{aligned}
&T_B(\tau_1,\tau_2) - T_A(\tau_1,\tau_2)\\
&\geq \sum_{k=b+W}^{e-W} D_B^k - \sum_{k=b}^{e} D_A^k\\
&= -\sum_{k=b}^{b+W} s_{max}^{k-1} - \sum_{k=e-W+1}^{w} s_{max}^{k-1} - s_B^{b-W-1} + s_B^{f-W} + s_A^{b-1} - s_A^e\\
&\geq -WL - WL - L_B + 0 + 0 - L_A\\
&\geq -2WL - L_A - L_B
\end{aligned}$$

That is, $|T_B(\tau_1, \tau_2) - T_A(\tau_1, \tau_2)| \leq 2WL + L_B + L_A$ still holds.

Combining both cases yields

$$|T_B(\tau_1,\tau_2) - T_A(\tau_1,\tau_2)| \leq 2WL + L_B + L_A$$

.

To compute $|T_B(t_1, t_2) - T_A(t_1, t_2)|$, we note that at time t_1 requests from P_A^{b-1} has started consuming dominant resources and at time t_2 requests from P_A^{e+1} are still consuming dominant resource. By definition, we have

$$|T_B(t_1,\tau_1) - T_A(t_1,\tau_1)| \leq \max(\omega(A,B), \omega(B,A))$$

and

$$|T_B(\tau_2, t_2) - T_A(\tau_2, t_2)| \le \max(\omega(A, B), \omega(B, A))$$

. It follows that

$$\begin{aligned}
&|T_B(t_1, t_2) - T_A(t_1, t_2)| \\
&\le |T_B(t_1, \tau_1) - T_A(t_1, \tau_1)| + |T_B(\tau_1, \tau_2) - T_A(\tau_1, \tau_2)| \\
&+ |T_B(\tau_2, t_2) - T_A(\tau_2, t_2)| \\
&\le 2WL + L_A + L_B + 2\max(\omega(A, B), \omega(B, A))
\end{aligned}$$

□

We further note that $\omega(i, j)$ should be negligible with respect to a backlogged period of multiple rounds and $L = \max_i\{L_i\}$, we can simplify the bound above.

Corollary A.9.

$$|T_B(t_1, t_2) - T_A(t_1, t_2)| \le 2WL + 2L$$

B DERIVATION OF EXPECTED THREAD POOL SIZE

In this section, we prove that our worker pool manager's scheme will not lead to an unbounded number of workers being created as a result of remote content fetching. We assume:

(1) At peak load, Varnish processes requests at an expected Poisson rate of λ.
(2) The latency of each content fetching from origin is independent and exponentially distributed at rate μ.
(3) Content fetches are served in parallel (i.e., all parallel content fetches are served by different origin servers).
(4) Every request to Varnish has an independent cache hit probability of p_h and a cache miss probability of $p_m = 1 - p_h$.

Lemma B.1. *At peak load, workers will be added to the worker pool at a rate $\lambda p_m = \lambda_m$.*

Proof. This follows from the assumptions (1) and (4). □

Theorem B.2. *The expected number of additional workers spawned by Sandpaper's pool manager is $\frac{\lambda p_m}{\mu}$.*

Proof. Let k be the number of requests that are currently making remote fetches. Note that by the definition of Sandpaper's pool manager, k is also the maximum number of workers currently added to the worker pool. We can model the number of requests that are fetching remote content using a continuous-time Markov chain (more specifically, a birth-death process):

$$\begin{tikzcd}
0 \arrow[r, bend left, "\lambda_m"] & 1 \arrow[l, bend left, "\mu"] \arrow[r, bend left, "\lambda_m"] & \cdots \arrow[l, bend left, "2\mu"] \arrow[r, bend left, "\lambda_m"] & k \arrow[l, bend left, "k\mu"] \arrow[r, bend left, "\lambda_m"] & \cdots \arrow[l, bend left, "(k+1)\mu"]
\end{tikzcd}$$

The states correspond to the number of requests currently blocked while fetching content from an origin server. The arrows show the expected rate at which we transition between states. Because we assume fetches are served in parallel the expected completion rate for requests (bottom arrows) is $k\mu$, rather than simply μ. We derive the time-reversability equations for the CTMC above.

State	Time-reversability equation	Simplified
0	$\pi_0\lambda_m = \pi_1\mu$	$\pi_1 = \frac{\lambda_m}{\mu}\pi_0$
1	$\pi_1\lambda_m = \pi_2 2\mu$	$\pi_2 = \frac{1}{2!}\left(\frac{\lambda_m}{\mu}\right)^2 \pi_0$
2	$\pi_2\lambda_m = \pi_3 3\mu$	$\pi_3 = \frac{1}{3!}\left(\frac{\lambda_m}{\mu}\right)^3 \pi_0$
k	$\pi_k\lambda_m = \pi_{k-1}k\mu$	$\pi_k = \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k \pi_0$

We also know

$$1 = \sum_{k=0}^{\infty} \pi_k = \sum_{k=0}^{\infty} \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k \pi_0,$$

so

$$\pi_0 = \frac{1}{\sum_{k=0}^{\infty} \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k}.$$

Notice that the denominator above is the series for the exponential function with exponent $\frac{\lambda_m}{\mu}$, so we have $\pi_0 = e^{-\frac{\lambda_m}{\mu}}$. For each k then,

$$\pi_k = \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k e^{-\frac{\lambda_m}{\mu}},$$

so

$$\begin{aligned}
\mathbf{E}[K] &= \sum_{k=0}^{\infty} \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k e^{-\frac{\lambda_m}{\mu}} k \\
&= e^{-\frac{\lambda_m}{\mu}} \sum_{k=0}^{\infty} \frac{1}{k!}\left(\frac{\lambda_m}{\mu}\right)^k k \\
&= e^{-\frac{\lambda_m}{\mu}} \left(\frac{\lambda_m}{\mu}\right) e^{\frac{\lambda_m}{\mu}} \\
&= \frac{\lambda_m}{\mu} = \frac{\lambda p_m}{\mu}
\end{aligned}$$

□