General Interest

# The Queuing-First Approach for Tail Management of Interactive Services

**Amirhossein Mirhosseini and Thomas F. Wenisch**
University of Michigan

***Abstract*—Managing high-percentile tail latencies is key to designing user-facing cloud services. Rare system hiccups or unusual code paths make some requests take $10\times - 100\times$ longer than the average. Prior work seeks to reduce tail latency by trying to address primarily root causes of slow requests. However, often the bulk of requests comprising the tail are not these rare slow-to-execute requests. Rather, due to head-of-line blocking, most of the tail comprises requests *enqueued behind* slow-to-execute requests. Under high disparity service distributions, queuing effects drastically magnify the impact of rare system hiccups and can result in high tail latencies even under modest load. We demonstrate that improving the queuing behavior of a system often yields greater benefit than mitigating the individual system hiccups that increase service time tails. We suggest two general directions to improve system queuing behavior–*server pooling* and *common-case service acceleration*–and discuss circumstances where each is most beneficial.**

**Online data-intensive (OLDI)** services (e.g., web search) traverse terabytes of data with strict latency targets.[1] Managing high-percentile tail latencies is a key problem in designing such services. First, to guarantee user satisfaction, services must meet strict response time service-level objectives (SLOs), especially for tail latencies.[2,3] Second, such services typically communicate via fan-out patterns wherein datasets are "shared" across numerous "leaf" servers and their responses are aggregated before responding to the user. As such, overall latency is often dictated by the slowest leaves (i.e., the "tail at scale" effect[4]).

*Digital Object Identifier 10.1109/MM.2019.2897671*
*Date of current version 23 July 2019.*


Published by the IEEE Computer Society

High tail latencies arise from two effects. First, such applications' service time distributions include outlying requests that take much longer ($10\times$ – $100\times$ or more) than the mean.[5] Some requests may require exceptional processing time depending on their arguments (e.g., search engines[1,6]) or query types (e.g., sets versus gets in key-value stores[5,7]). Some requests are delayed by system interference, such as from garbage collection, page deduplication, synchronous huge-page compaction, or network stack impediments.[4,8] In other cases, scheduler inefficiencies, power state transitions, suboptimal interrupt routing, poor NUMA node allocation, or virtualization effects may contribute to long tail latencies.[9] Finally, interference from colocated workloads can cause slowdown due to contention for shared caches, memory bandwidth, or global resources like network cards or switches.[10,11]

A second key contributor to applications' end-to-end latency distribution are queuing effects.[3] Queuing arises at numerous layers causing some requests to wait for others[4]; whereas queuing also affects average performance, its effect on tail latency may be catastrophic. To achieve performance stability, systems must be engineered such that the overall request arrival rate is lower than the aggregate system capacity (service rate). However, as both rates fluctuate, arrivals may temporarily outstrip service capacity, causing requests to queue. Queueing delay is most apparent under high system load. However, in this paper, we make the case that queuing effects drastically magnify the impact of rare system events/hiccups and can result in high tail latencies even under modest load. Due to head-of-line (HoL) blocking, many requests are delayed by an exceptionally slow one that stalls a server/core; these delayed requests account for a bulk of the latency distribution tail.

Online data-intensive (OLDI) services (e.g., web search) traverse terabytes of data with strict latency targets. Managing high-percentile tail latencies is a key problem in designing such services.

Through stochastic queuing simulation,[12] we show that improving a system's queuing behavior often yields much greater benefit than mitigating the individual system hiccups that increase service time tails. We suggest two general directions for improving system queuing behavior: server pooling, and common-case service acceleration (CCSA). Server pooling is the practice of redesigning system architecture to change single-server ("scale-out") queues into multiserver ("scale-up") ones; that is, rather than enqueuing requests at distinct servers/cores, a single queue is shared among many (i.e., converting $c$ G/G/1 queues into a G/G/c). Server pooling greatly reduces queuing delay and can completely eliminate queueing with enough servers (i.e., high enough $c$). Pooling smooths fluctuations in both arrivals and service, making the system behave more like one with deterministic interarrival and service times. Especially for high disparity service time distributions (i.e., rare system events/hiccups), server pooling reduces the overall tail latency by breaking HoL blocking and preventing nominal requests from waiting behind exceptionally long ones. Even a modest degree of concurrency allows many short requests to drain past stalled ones, substantially reducing weight in the latency distribution tail.

CCSA improves systems' queuing behavior by deploying optimizations that target common-case service behavior (as opposed to optimizations that target directly rare/slow requests or hiccups). It may seem counterintuitive to improve *tail* latency by optimizing *typical-case* request performance, but queuing delays are greatly impacted by the average load, which depends more on typical-case service time than rare cases.

In single-server systems, CCSA has little impact when the service variance is excessively high (i.e., HoL blocking is common), as nominal requests queue behind rare, slow ones regardless of how fast the nominal requests are processed. But, if there is sufficient concurrency (e.g., by using server pooling) that slow requests rarely occupy all servers, then CCSA provides enormous benefit by allowing nominal requests to drain past slow ones, drastically reducing wait time. Importantly, we show that, with concurrency, CCSA is more effective than reducing directly either the length or the probability of rare hiccups. Since finding and mitigating tail events is hard due to their myriad causes,[13] we

believe this observation is encouraging—we can reduce tail latency without engaging in "whack-a-mole" with rare system hiccups.

In short, we argue that cloud system designers should invest optimization effort first into 1) reducing HoL blocking through higher concurrency and improved queuing discipline (i.e., server pooling) and then into 2) optimizing common-case performance to improve mean service time. Both of these approaches may have greater impact and are easier to achieve than directly pinpointing and mitigating rare cases and hiccups; whereas server pooling smooths out arrival and service variability, CCSA reduces the effective system load. The relative impact of the two approaches depends critically on the system load and service time variance. CCSA's effectiveness improves as service times become more normal and/or concurrency increases. We build a simple regression model on concurrency and service time variance to estimate HoL blocking and indicate whether server pooling or CCSA is more beneficial in reducing tail latency. System designers can use this model to guide optimization effort and estimate its impact.

Neither inter-arrival nor service times of interactive cloud services are perfectly modeled by exponential distributions. But since requests usually originate from a large pool of independent sources (e.g., many distinct users), they typically mimic Poisson (memoryless) arrivals.

## BACKGROUND AND METHODOLOGY

Most interactive cloud services can be modeled as $A/S/c$ queuing systems (based on Kendall's notation[14]), where $A$ specifies the request inter-arrival time distribution, $S$ the service time distribution, and $c$ the number of concurrent servers. Regardless of the distributions, the average arrival rate ($\lambda$) must be lower than the average aggregate service rate of all servers ($\mu c$, with $\mu$ as the average service rate of a single server); otherwise, requests queue without bound.

The most common queuing models used in analytical studies are $M/M/c$ systems ($M$ stands for Markovian), where both interarrival and service times follow exponential distributions. It can be shown that the exponential distribution is the only continuous distribution with the memoryless property (i.e., occurrence of events is independent of the system's history).[14] An interesting property of exponentially distributed random variables is the constant ratio between their mean values and all of their quantiles (including the median and all-percentile tails), as shown in

$$P(S > \alpha E(S)) = e^{-\alpha}. \quad (1)$$

Due to the memoryless property of exponential distributions, $M/M/c$ queuing systems can be easily analyzed with continuous time Markov chains and have closed-form solutions for many of their parameters, such as average waiting and sojourn (waiting plus service) times.

Neither inter-arrival nor service times of interactive cloud services are perfectly modeled by exponential distributions. But since requests usually originate from a large pool of independent sources (e.g., many distinct users), they typically mimic Poisson (memoryless) arrivals; prior studies have observed that interarrival time distributions usually have small coefficients of variation (mostly, between 1 and 2).[12] As such, interarrival processes can be well approximated with an exponential distribution ($CV = 1$) with little fidelity loss.[15] Service time distributions, in contrast, may have long tails; some requests encounter rare hiccups that increase service time by $10\times - 100\times$ (or even more) over the mean—much larger than the ratio of the 99th percentile and mean values in the exponentially distributed services times of $M/M/c$ systems [$\sim 4.6$, based on (1)]. Hence, interactive cloud services are often investigated using $M/G/c$ queuing models ($G$ stands for General).[5,16]

Unfortunately, $M/G/c$ queuing models do not have closed-form solutions for average waiting/sojourn times and the accuracy of existing approximations, which use only a few moments, is poor.[17] Furthermore, to the best of our knowledge, there is no widely used approximation for waiting/sojourn time quantiles of these systems. Thus, we use stochastic queuing simulation, based on the BigHouse methodology,[12] to measure the tail latency of such $M/G/c$ systems. We

simulate the queuing system until we achieve 95% confidence intervals of 5% error in reported results. We consider the first-come–first-served (FCFS) queuing discipline as prior work[9,18] shows it to be the best non-preemptive scheduling policy when tail latency is the metric of interest.

We model "nominal" request performance by drawing service times from an exponential distribution with mean $1/\mu_n$. Then, to represent rare/slow requests, which we call "hiccups," with probability $p_h$, we add an additional delay drawn from a second exponential distribution with mean $1/\mu_h$. We vary both $p_h$ and the ratio of $\mu_n/\mu_h$ in our experiments. This hybrid model is similar to the dual-branch Hyperexponential distribution, which is widely used as a phase-type distribution for approximating heavy-tailed systems.[14] We study analytical distributions as they are easier to understand and their parameters can be tuned to model various real scenarios.

The intent of our approach is to model the near-memoryless nominal behavior of cloud services and then overlay an independent distribution to model hiccups. We consider hiccups that are 1) $10\times$ longer than average, affecting 1% of requests, and 2) $100\times$ longer affecting 0.1% of requests.

1) The first one represents unusual code paths that arise in, e.g., web search. As an example, Microsoft observes a bimodal distribution for Bing search,[6] wherein most requests incur latencies close to the mean but occasional requests require an order of magnitude more processing time due to their complicated search queries. They report a $27\times$ ratio between the 99th percentile tail and the median latency (which is usually smaller than the mean). Similarly, Google reports a 1-ms median leaf service time with 99th percentile tail latency of 10 ms.[4]
2) This one represents rare pauses that arise due to system activities and interference. As an example, Wang *et al.*[19] studied a multitier web application and identified a similar bimodal distribution incorporating rare requests with less than 1% probability that take $30-40\times$ longer than the mean due to "transient events," such as JVM garbage collection or voltage/frequency state transitions.

## THE QUEUEING-FIRST APPROACH

Requests may incur an end-to-end latency in a high percentile tail either because the request itself incurred a rare hiccup or due to queuing delays. Queuing greatly magnifies the impact of few, rare hiccups by causing nominal requests to queue behind one with a hiccup and incur high sojourn times. With deterministic or memory-less service times, queuing arises primarily due to request bursts, wherein the instantaneous arrival rate exceeds the average service rate. However, with high disparity service time distributions, queuing is caused mostly by HoL blocking, wherein the instantaneous service rate drops temporarily well below the average request arrival rate.

The differing nature of queuing has important implications. First, with high disparity service, queuing can arise even at low load; when a slow request stalls the server for a long time, many requests may queue behind it, even if the arrival rate is low. Second, it increases the contribution of nominal requests to the sojourn-time tail; while hiccups directly impact few requests, such requests account for a large fraction of server utilization. As such, a substantial fraction of nominal requests queue behind the exceptional ones. As an example, in an $M/G/1$ queue where 0.1% of requests incur a $100\times$ higher than nominal service time, the exceptional requests account for $\sim$10% server utilization. As a consequence of Poisson arrivals, $\sim$10% of requests arrive during such a slow service and may also contribute to the sojourn-time tail.

Figure 1(a) and (b) reports the normalized 99th percentile tail latency of an $M/M/1$ system and its $M/G/1$ counterpart with the high disparity service time distribution described above across various load levels. Figure 1(c) reports the fraction of sojourn time spent waiting by the 1% slowest requests for both $M/M/1$ and $M/G/1$ queues. Under low loads, wait time is usually small in $M/M/1$ systems and the sojourn-time tail is nearly the same as the service-time tail. However, queuing accounts for a significant fraction of tail latency when the

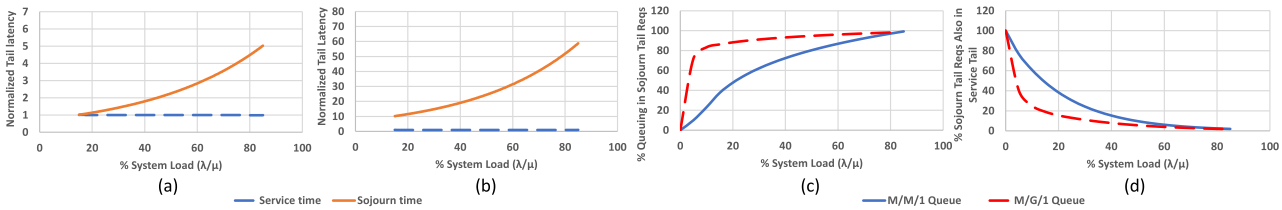


**Figure 1.** (a) Normalized service- and sojourn-time 99th percentile tail in an M/M/1 queue. (b) Normalized service- and sojourn-time 99th percentile tail in an M/G/1 queue. (c) Average % wait time in sojourn-time tail requests. (d) % of sojourn-time tail requests that are also in the service-time tail. The M/G/1 queue has an exponential service time distribution, but incorporates $100\times$ hiccups that occur in 0.1% of the requests.

service time distribution is high disparity. Furthermore, since hiccups occur with a low probability (0.1%), they do not noticeably affect the service time 99th percentile tail. However, due to the HoL blocking, their impact on the sojourn-time tail is large under both low and high loads.

Figure 1(d) reports the percentage of requests in the sojourn-time tail that also contribute to the service-time tail. Under both low and high loads, the percentage is much higher in the $M/M/1$ system. With high disparity service times, HoL blocking in the $M/G/1$ system comprises the bulk of the tail—most sojourn-time tail requests are nominal requests that queue behind exceptionally slow ones. Furthermore, as shown in Figure 1 (c), while the fraction of queuing delay relative to sojourn time in tail requests is higher in the $M/G/1$ system, queuing still accounts for more than half of sojourn time even in $M/M/1$ systems for loads over $\sim$30%.

The takeaway is that if a service incurs either high load or has a high disparity service time distribution, end-to-end tail latency is dominated by queuing effects. As a result, improving system queuing behavior is typically more effective than seeking to directly mitigate system hiccups that cause heavy/long tails. Finding and mitigating system hiccups is hard. As such, we advocate pursuing optimizations that address queuing behavior instead.

## Server Pooling

Figure 2 contrasts two different models to compose multiple servers. In the scale-out model, each server has a separate request queue and a dispatcher/load balancer steers incoming requests into different queues such that the request arrival rate of all servers is balanced. In the scale-up model, instead a single request queue is shared among all servers, which each fetch requests from the central request queue as they become idle. This model requires synchronization of the central request queue, but improves queuing.

It can be shown that the scale-up ($M/G/c$) organization always outperforms the scale-out organization ($c - M/G/1$) in principle (neglecting synchronization). First, in the scale-up organization, a server will not remain idle if there are requests waiting in the central queue. However, in scale-out systems, a server may remain idle if its own queue is empty even while other servers have outstanding requests. Second, when a request takes longer than average in a scale-out organization, all the requests behind it suffer from

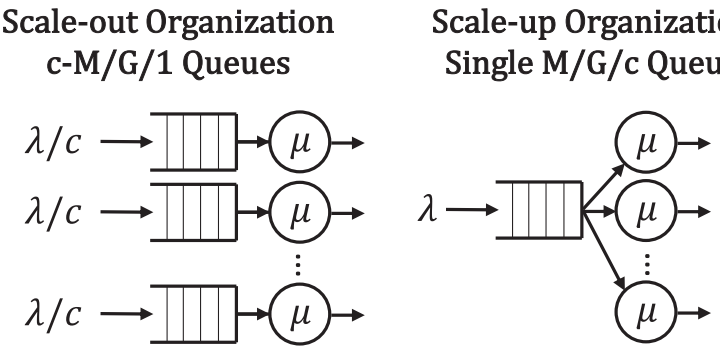


**Figure 2.** Scale-out versus scale-up queuing organizations.

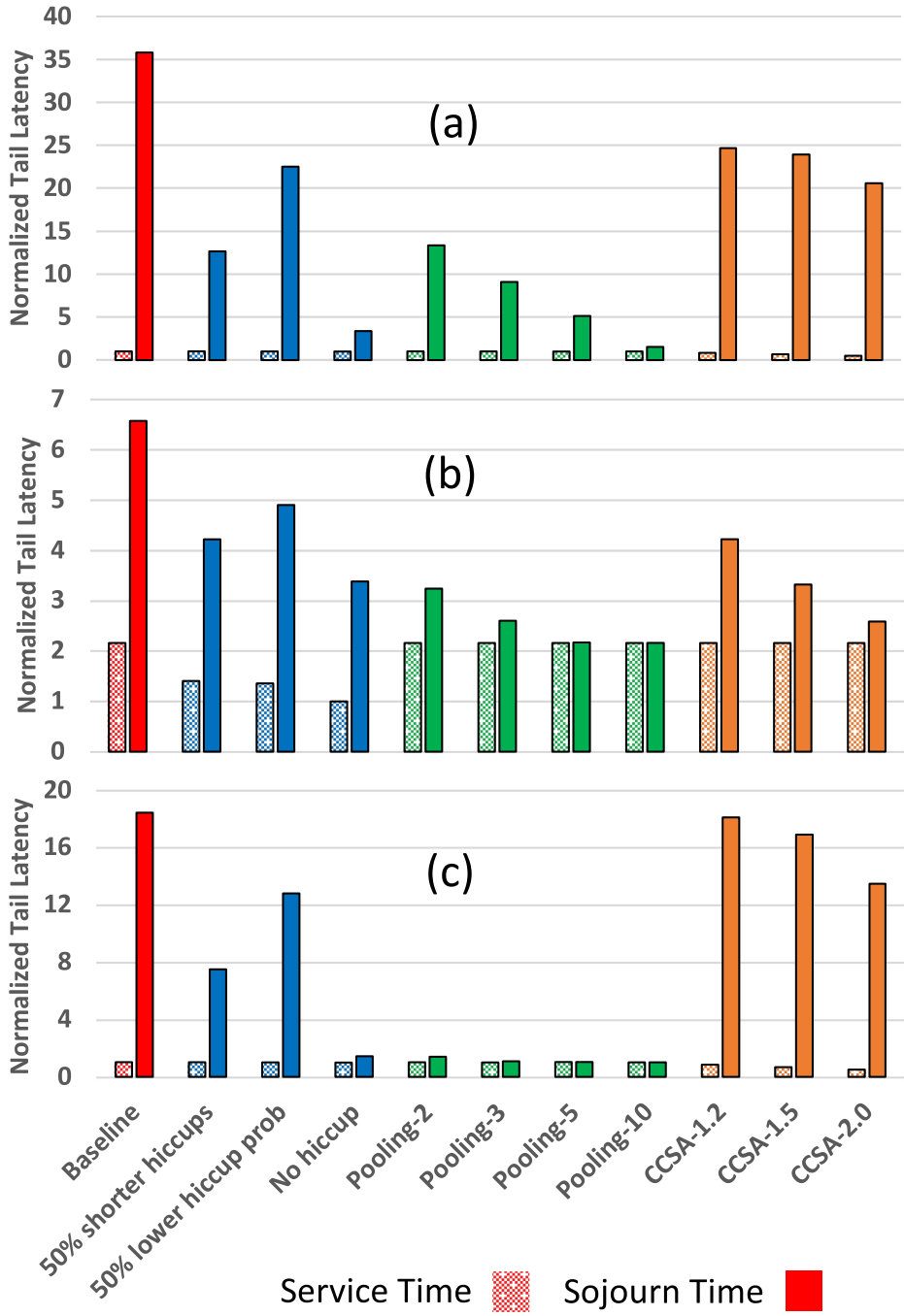


**Figure 3.** Normalized service-time (light bars) and sojourn time (dark bars) tails of an $M/G/1$ queue under different scenarios. (a) 70% load, $100\times$ hiccups affecting 0.1% of requests. (b) 70% load, $10\times$ hiccups affecting 1% of requests. (c) 30% load, $100\times$ hiccups affecting 0.1% of requests.

HoL blocking delays. In contrast, in scale-up architectures, requests may be serviced by any server; stalling at one server has 1little impact on system-wide instantaneous service rate.

Several prior studies have observed that scale-up queuing systems outperform scale-out organizations.[9,5] However, a large number of contemporary software systems use a scale-out queuing architecture as it is easier to implement.[9] Implementing a scale-up model across multiple machines requires remote disaggregated memory accesses or a distributed data structure, which are difficult to implement and optimize. Even within a single

multicore server, implementing a scale-up model mandates either a single synchronized data structure or a work-stealing architecture, which incur coherence traffic and are difficult to scale.

We refer to the practice of consolidating $c - M/G/1$ servers into a single $M/G/c$ system as server pooling. When service-time distributions are high disparity, HoL blocking becomes the main source of queuing delay (and tail latency) and the gap between the two queuing organizations grows. We argue that server pooling can play a key role in resolving HoL blocking under such service conditions and, hence, should be pursued despite higher implementation complexity. In fact, server pooling often reduces the tail latency more than directly mitigating the rare hiccups that cause exceptionally long service.

Figure 3 reports the normalized service/sojourn time tail latencies in an $M/G/1$ system with different service time distributions and system loads. The leftmost red bars represent tail latencies in the presence of rare hiccups. The next group of blue bars show the tail latency where the impact (i.e., duration/probability) of hiccups has been reduced. In particular, from left to right, these bars represent cases where hiccup duration is halved, their occurrence probability is halved, and where hiccups are fully eliminated. Finally, the cluster of green bars indicate server pooling cases with varying number of servers $c$. (We discuss the orange bars later.)

Figure 3(a) considers an exponential service time distribution with hiccups that occur 0.1% of the time and last $100\times$ longer than the average service time under 70% system load. We make three observations: First, reducing the hiccup probability is considerably less effective at reducing the overall tail than reducing their duration. The intuition is that longer hiccups cause more requests to queue and hence exacerbate tails more than shorter but more frequent hiccups. Second, pooling only two servers reduces tail latency almost as much as halving hiccup durations. Whereas it may be challenging to implement high-concurrency data structures to enable a high degree of server pooling, sharing queues across just pairs of machines or cores is likely easier than finding and mitigating hiccups. Finally, with greater degrees of server pooling, queuing delay vanishes and the sojourn-time tail and service-time tail match. In such a scenario, end-to-end tail latency is even lower than in a system with no hiccups but without pooling.

Figure 3(b) reports the same results for hiccups $10\times$ longer than the average occurring in 1% of requests. Whereas the general trend matches Figure 3(a), the gap between the service- and sojourn-time tails is noticeably smaller even though the total service time attributable to hiccups is the same ($10 \times 1\% = 100 \times 0.1\%$). As previously observed, longer hiccups introduce more severe HoL blocking and cause more nominal requests to queue behind the exceptional ones (despite lower hiccup probability). Nevertheless, in Figure 3(b), pooling across only two servers, despite hiccups, is enough to reduce the sojourn time tail below that of a system without server pooling and without hiccups. Figure 3(c) considers the same service time distribution as Figure 3(a), but under lower (30%) system load. Here, whereas queuing delays are typically near-negligible under low load, the high disparity service distribution nevertheless causes HoL blocking and a significant sojourn time tail. Interestingly, the ratio between the sojourn- and service-time tails is much higher than that seen in Figure 3(b) due to longer hiccups and higher HoL blocking, despite lower load. Furthermore, when HoL blocking is high but system load is low, pooling across two servers completely eliminates queuing delay.

In summary, server pooling is highly effective in eliminating HoL blocking and reducing queuing delays that otherwise arise due to rare system hiccups. Although pooling across many cores/machines is often challenging, encouragingly, we show that pooling across as few as two servers is often sufficient for large tail latency reductions.

A variety of steering and scheduling techniques can enable a scale-out system to more closely approximate scale-up system behavior. Examples include smart load-balancing schemes that steer requests to queues based on wait time estimates derived from metrics like queue occupancy, injecting replica requests to different queues and then cancelling the redundant requests,[4] and various work-stealing approaches

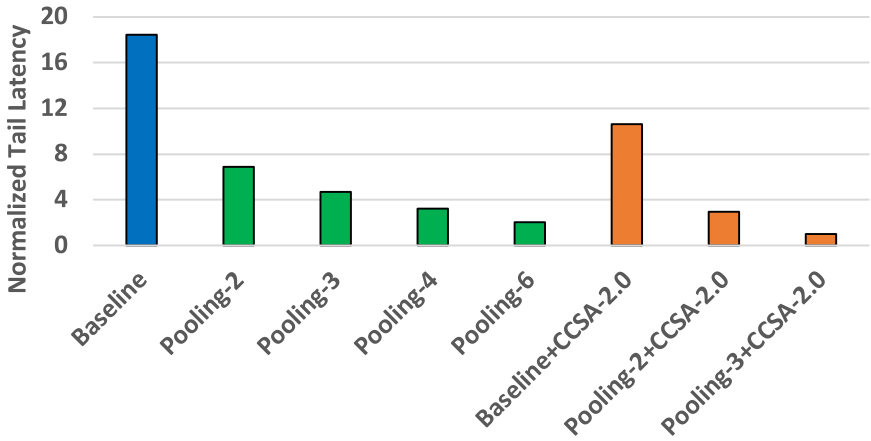


**Figure 4.** Normalized sojourn time tail latency in an $M/G/1$ queue ($100\times$ hiccups in 0.1% of requests) with various degrees of server pooling and CCSA.

that migrate tasks between queues.[20] While these techniques typically fall short of an ideal $M/G/c$ system, they still drastically reduce wait time and HoL blocking. Further, as shown by Wierman and Zwart,[18] FCFS scheduling is only best for $M/G/1$ queues if the service-time distribution is light-tailed. Otherwise, variants of processor sharing outperform FCFS in terms of tail latency. Thus, should direct implementation of server pooling prove prohibitive in a particular system, time-multiplexing machines/cores among requests may provide an alternative to address queuing due to rare hiccups.

### Common-Case Service Acceleration

CCSA is another general approach to improve system queuing behavior. In this approach, rather than seek to mitigate the rare hiccups that cause high service times, instead, the system designer deploys optimizations that accelerate *common case* behavior. As such, while CCSA directly reduces average service time, it has little effect on the tail of the service distribution. Conventional wisdom suggests that improving average service time does not improve tail latency; indeed, some prior work suggests trading off slower average performance to reign in tails.[7,6] However, reducing average service time increases service rate, and hence reduces server utilization. Reduced utilization in turn reduces queuing delays.

Unlike server pooling, CCSA has little impact when HoL blocking is high, as nominal requests enqueue behind long ones regardless of how fast nominal requests are processed. The rightmost set of orange bars in Figure 3 report service and sojourn time tails under varying degrees of CCSA (i.e., different speedups of common-case service time). We observe a large benefit in Figure 3 (b), where hiccups are relatively short and there is little HoL blocking; accelerating the common-case service time by only 20% (without affecting its tail) reduces the sojourn time tail almost as much as reducing the average hiccup length by half. Doubling the service rate reduces the sojourn time tail below that of a system with no hiccups or a system with a pooling degree of two. In the remaining cases [see Figure 3(a) and (c)], CCSA has only modest impact on the sojourn time tail, as there is more HoL blocking and insufficient concurrency for requests to avoid it

CCSA provides greater benefit with higher concurrency (e.g., via server pooling). Even modest concurrency is sufficient to unlock CCSA's effectiveness; rare events are unlikely to occupy multiple servers at the same time, so nominal requests nearly always bypass a stalled server. As an example, Figure 4 considers the scenario from Figure 3(a) but in $M/G/2$ and $M/G/3$ systems (i.e., with server pooling). Not only is CCSA better than mitigating hiccups, it is also better than further increasing server pooling. We expect that CCSA will typically be easier to implement than hunting down and optimizing the underlying causes of rare performance hiccups as software developers are already incentivized to make the common case fast. Note that this approach is most beneficial for service distributions wherein, despite heavy/long tails, most of the system utilization arises from nominal requests. For example, in our modeled distributions, $\sim$10% of system utilization is spent on hiccups. However, in many power-law distributions, tail events contribute to 80%–90% of the distribution; CCSA would not be as effective in such scenarios.

### Discussion

We expect CCSA to be more beneficial than server pooling, as CCSA reduces the effective system load, while server pooling has no effect on

load/utilization. However, as we showed in the previous section, CCSA is only effective in the absence of server HoL blocking. When HoL blocking is frequent (e.g., service time variance is high), CCSA no longer provides benefit as nominal requests queue behind exceptionally long ones. In such scenarios, additional concurrency must be introduced to unleash CCSA's efficacy.

In single-server systems, service time variability is a good measure of HoL blocking. For example, in Figure 3(a), where $CV_{\text{service}} = 4.2$, CCSA has negligible impact; nominal requests wait behind slow ones. In contrast, in Figure 3(b) where $CV_{\text{service}} = 1.6$ (near the $CV_{\text{service}} = 1.0$ of $M/M/*$ queues), CCSA is more effective than server pooling. However, $CV_{\text{service}}$ only reflects HoL blocking in single-server systems. We suggest the interdeparture time variability of a saturated queue (when queuing probability is close to 1.0) to measure HoL blocking in multi-server queues. In saturated single-server queues, the interdeparture time distribution is the service time distribution ($CV_{\text{service}} = CV_{\text{departure}}$). However, with multiple servers, departures interleave, reducing interdeparture time variability. For example, in Figure 4, which is similar to Figure 3(a) but with an additional 1–2 servers, the $CV_{\text{departure}}$ drops (from 3.0) to 1.7 and 1.1, respectively. As a result, in the $M/G/2$ case, CCSA yields almost the same benefit as pooling. In the $M/G/3$ case, where HoL blocking resembles that of an $M/M/*$ queue with $CV_{\text{departure}} = 1.0$, CCSA yields much better results than server pooling.

We find that a simple regression model can predict the $CV_{\text{departure}}$ of a saturated $M/G/c$ queue based on its $CV_{\text{service}}$ and the number of servers ($c$). We construct the model by simulating saturated queues with a set of high disparity distributions with different $CV_{\text{service}}$ and measure their $CV_{\text{departure}}$. We observe that small degrees of server pooling quickly reduce HoL blocking. Therefore, we postulate an exponential decay effect for the number of servers. Also, we note that $CV_{\text{departure}}$ may not decrease below 1.0 as the interdeparture process becomes near-memoryless around $CV_{\text{departure}} = 1.0$, where the ratio of tail-to-average cases does not decrease through higher concurrency [see (1)]. As a result, we suggest a regression model of the form of

$$CV_{\text{departure}} \approx (CV_{\text{service}} - 1)e^{-0.8(c-1)} + 1 \quad (2)$$

and tune its parameter using the least squares method. We find its average error to be less than 13%.

Using this model, we can derive $CV_{\text{departure}}$ as a proxy for the HoL blocking rate and predict how it is affected by server pooling. Alternatively, cloud system architects may perform Stochastic Queueing Simulations, similar to our approach, and directly measure $CV_{\text{departure}}$ instead of predicting it. When the system approaches the $CV_{\text{departure}} = 1.0$ of $M/M/*$ queues, blocking becomes rare; the remaining tail of the sojourn time distribution is then primarily due to service time tails or high load. Under low load, queuing delays vanish with sufficient server pooling; remaining sojourn time tails reflect only service tails. Under high load, HoL blocking will no longer be the dominant source of queuing delays when sufficient concurrency has been introduced. As such, with sufficient server pooling (often just 2–3 servers), CCSA becomes more effective than further server pooling.

We recommend developers follow a simple optimization sequence to address tail latency in their services: 1) introduce server pooling until HoL blocking is sufficiently mitigated; 2) if load is high, introduce CCSA; 3) if end-to-end tails remain unacceptable, only then seek to directly optimize rare, high service latencies.

In short, we recommend developers follow a simple optimization sequence to address tail latency in their services: 1) introduce server pooling until HoL blocking is sufficiently mitigated; 2) if load is high, introduce CCSA; 3) if end-to-end tails remain unacceptable, only then seek to directly optimize rare, high service latencies.

## CONCLUSION

Improving a system's queuing behavior often yields much greater benefit than mitigating the individual system hiccups that increase service time tails. We suggest two general directions for improving system queuing behavior—server pooling, and CCSA—which synergistically address queuing behaviors that often drive tail latency.

## ACKNOWLEDGMENTS

This work was supported by the Center for Applications Driving Architectures, one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

## REFERENCES

1. L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar./Apr. 2003.
2. S. Kanev *et al.*, "Profiling a warehouse-scale computer," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 158–169.
3. C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Commun. ACM*, vol. 61, no. 8, pp. 65–72, 2018.
4. J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
5. H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE Int. Symp. Workload Characterization.*, 2016, pp. 1–10.
6. M. E. Haque *et al.*, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 161–175, 2015.
7. C.-H. Hsu *et al.*, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 271–282.
8. R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, Art. no. 9.
9. J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
10. D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proc. ACM SIGARCH Comput. Archit. News*, 2015, vol. 43, no. 3, pp. 450–462.
11. A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *Proc. IEEE 25th Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 185–198.
12. D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2012, pp. 35–45.
13. X. Yang, S. M. Blackburn, and K. S. McKinley, "Computer performance microscopy with shim," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 170–184.
14. M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge, U.K.: Cambridge Univ. Press, 2013.
15. D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 319–330.
16. D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 205–216, 2009.
17. V. Gupta, M. Harchol-Balter, J. Dai, and B. Zwart, "On the inapproximability of m/g/k: Why two moments of job size distribution are not enough," *Queueing Syst.*, vol. 64, no. 1, pp. 5–48, 2010.
18. A. Wierman and B. Zwart, "Is tail-optimal scheduling possible?" *Oper. Res.*, vol. 60, no. 5, pp. 1249–1257, 2012.
19. Q. Wang *et al.*, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 31–40.
20. J. Li *et al.*, "Work stealing for interactive services to meet target latency," *ACM SIGPLAN Notices*, vol. 51, no. 8, p. 14, 2016.

**Amirhossein Mirhosseini** is currently working toward a PhD in computer science and engineering at the University of Michigan. His research interests center on computer architecture with particular emphasis on data centers, cloud computing, and microservices. He is a student member of the IEEE and the Association for Computing Machinery (ACM). Contact him at miramir@umich.edu.

**Thomas F. Wenisch** is an associate professor of electrical engineering and computer science and Associate Chair of External Affairs at the University of Michigan, Ann Arbor. His research interests center on computer architecture with particular emphasis on server and data center systems, memory persistency, multiprocessor systems, and performance evaluation methodology. He has a PhD in electrical and computer engineering from Carnegie Mellon University. He is a member of the IEEE and the Association for Computing Machinery (ACM). Contact him at twenisch@umich.edu.