Leveraging Queueing Theory and OS Profiling to Reduce Application Latency

Anshul Gandhi, Amoghvarsha Suresh
PACE Lab, Stony Brook University
{amsuresh,anshul}@cs.stonybrook.edu

Abstract

Request latency is a critical metric in determining the usability of online services, such as web applications and databases. Most existing approaches to improve application latency start by detecting bottlenecks in the application deployment; this typically entails determining stages of processing where the application spends most of its time. Inspired by queueing theory, we present an alternative approach to detect and mitigate bottlenecks âĂŞ using variability of processing time as a guiding principle when designing applications.

CCS Concepts

• **Computer systems organization** → *Client-server architectures.*

ACM Reference Format:
Anshul Gandhi, Amoghvarsha Suresh. 2019. Leveraging Queueing Theory and OS Profiling to Reduce Application Latency. In *20th International Middleware Conference Tutorials (Middleware Tutorials '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3366625.3368853

Introduction

Web applications provide important services, such as online retail, messaging, and search, to end-users on a daily basis [1–3, 5]. A critical metric for determining the usability of such web applications is the *latency* of user requests; different service providers employ different measures of latency, such as mean, median, 95%ile, and 99%ile [4, 17, 22]. A delay of even a few milliseconds in request latency can lead to significant revenue loss for service providers due to user abandonment [19, 26].

Conceptually, request latency can be broken down into two distinct components, *service time* and *waiting time* [27]. Service time is defined as the time during which the request is being actively serviced. Waiting time is then defined as the remaining time during which the request is waiting to be served. To maintain acceptable latencies, service providers often optimize their web servers to reduce the mean service time of requests, that is, shorten the critical path of request processing. For example, Chronos [12] uses user-level networking with NIC-level request dispatch to reduce lock contention and lower the latency of web applications; Jose et al. [24] make Memcached RDMA-capable to shorten the critical path; Li et al. [15] advocate using a real-time scheduler to reduce request scheduling delay.

An alternative approach to reducing web latencies that we explore in this tutorial is to minimize the *variability* in request processing. Queuing models show that, in addition to mean service time, the variability of the service time is also important when trying to reduce latency. Surprisingly, there has been very little work on actually reducing the variability in the system [4, 10, 23]. While much of the variability is intrinsic to the workload (e.g., burstiness in customer traffic), some of the variability is due to the *application and software design* (e.g., garbage collection, context switches, CPU scheduling policies, etc.), and can be regulated by making subtle changes to the system.

In this work, we investigate the following system design question – "*is it worth reducing variability in request processing times at the potential expense of lengthening its critical path?*". While theoretical analysis suggests that this is indeed the case, especially for heavy-tailed distributions (see Section 3), evaluating this idea in practice for web applications is challenging for several reasons:

- Web applications have a *complex processing lifetime*, going through several paths of processing in the kernel, including the TCP stack, parsing of requests, and scheduling of the request on server cores. Identifying the most likely culprit(s) that contributes to service time variability will require low-overhead yet accurate and fine-grained *request tracing* in the user and kernel space.
- There are several control knobs within a web server that can be tuned to reduce service time variability, such as the OS scheduler, page allocation strategy, etc. Prior work has also shown that new application-specific control knobs can be dynamically generated [7]. Given the numerous choices, efficiently finding the right control knob to mitigate variability is challenging, especially since the choice of the optimal control knob may depend on the server and web application configuration. Worse, employing the wrong control knob can *hurt* request latency.
- Most control knobs within the system that can be tuned to reduce service time variability *invariably hurt mean service time.* For example, prior work has shown that admission control can reduce service time variability by preventing server overload [11]. But this reduction comes at the expense of lengthening the critical path (since requests are held back), and possibly hurting latency. It is thus important to carefully tune the control knob to balance the trade-off between variability and mean service time.

We address the above challenges in the context of web services and demonstrate the benefits of *using service time variability as a guiding principle* to improve request latency. We employ light-weight, fine-grained request profiling to track service time variability; this allows us to identify components on the critical path of the request-response cycle that exacerbate service time variability. Using variability as our guiding principle, we then determine control knobs in the system and/or application that can be tuned to mitigate service time variability in the identified components. We evaluate our approach by investigating variability in Memcached, a popular in-memory (key-value store) caching service used to speed up web applications [25], and the Apache web server [29], a widely deployed http server application.

Middleware Tutorials '19, December 9–13, 2019, Davis, CA, USA
2019. ACM ISBN 978-1-4503-7040-0/19/12…$15.00
https://doi.org/10.1145/3366625.3368853

Tutorial Overview

The objective in this tutorial is to demonstrate the applicability of queuing models to reducing latency in web applications. We will start with a queueing theory primer, focusing on how to employ a queueing model for real-world applications. In particular, we will focus on key lessons learned from queueing theory and how these lessons can be applied to real-world settings. We will then explain how we use these lessons, along with OS profiling, to detect and mitigate bottlenecks in online applications. Our approach will focus on measuring the variability in processing times of a request as it traverses software layers within the user and kernel space, and then identifying the most critical stages of request processing. We will then present three use cases to illustrate our approach and its latency improvement benefits: (i) a Memcached server (two different settings), and (ii) the Apache web server. We will conclude with thoughts on how the approach can be extended to other applications, including emerging computing paradigms, such as microservice architectures.

List of Topics:

- Basics of queueing theory: arrivals, departures, queues
- Queueing models: M/M/1, M/M/k, M/G/1, M/G/k
- Useful lessons: latency vs. load, heavy-tail distributions, impact of variability
- Shortcomings: limiting assumptions, practical applicability
- Using queueing theory to detect application bottlenecks
- Application profiling: OS-level profiling, service time, stages, root cause analysis
- Finding the right control knobs in the OS and the application
- Case studies: Memcached, Apache web server

Background and Motivation

To motivate our approach of focusing on service time variability, we leverage queueing theory to analyze request latency for a server as a function of variability. Although models are only approximations of today's complex applications, the resulting analysis is instructive [9, 14, 16, 21], and guides our system design in later sections.

Request latency versus variability

Recent studies at Bing [8, 10], Google [12, 13, 18], and Facebook [1], suggest that modern web applications often experience high variability in inter-arrival time (IAT) and service time (ST). Variability in IAT represents workload variability, such as bursty arrivals. ST variability represents variability in processing times due to differences in work requirements (e.g., reads vs. writes) or differences in the request path (e.g., due to batching or TLB misses), or due to misconfigurations at the server that lead to anomalous behavior or "jitters" [15]. Note that ST is the amount of service required by a request to complete processing; alternatively, ST is the minimum possible request latency, assuming no delays.

To investigate the impact of variability on request latency, we consider a web server with a given inter-arrival time (IAT) distribution and a given service time (ST) distribution. To parameterize variability, we use the squared coefficient of variation (C^2), defined as the ratio of variance and square of the mean. Then, we have:

$$C^2_{IAT} = Var(IAT)/E^2[IAT], \text{ and} \quad (1)$$

$$C^2_{ST} = Var(ST)/E^2[ST], \quad (2)$$

where $E[]$ is the mean and $Var()$ is the variance of the random variable. To examine the full range of variability ($[0, \infty)$), we consider the following distributions:

- D (Deterministic), with $C^2 = 0$, is the ideal case of no variability.
- M (Exponential), with $C^2 = 1$, represents nominal variability.
- H_2 (Hyper-exponential), with $C^2 > 1$ (customizable), represents the case of high variability.

The $M/G/1$ queueing model, with Exponential IAT and generic ST distributions, allows us to analyze mean request latency, $E[T]$, as a function of ST variability, $Var(ST)$, and mean ST, $E[ST]$, via the Pollaczek-Khinchin (P-K) formula [27]:

$$E[T] = Var(ST) \cdot \frac{\lambda}{2(1-\rho)} + E[ST] \cdot \frac{2-\rho}{2(1-\rho)}, \quad (3)$$

where $\lambda = 1/E[IAT]$ is the mean request arrival rate and $\rho = \lambda \cdot E[ST]$ is the normalized system load [27]. Clearly, both $E[T]$ *and* $Var(ST)$ impact request latency.

Consider the $M/G/1$ model where all parameters, including $E[ST]$, are fixed, and only $Var(ST)$ is varied. Let the workload have a mean ST, $E[ST]$, of 1 millisecond, and system load of, say, 60%; thus, request rate, $\lambda = 0.6 \cdot E[ST] = 600$ req/s. Using Eq. (3), we find that $E[T] = 1.7ms$ under Deterministic ST ($C^2_{ST} = 0$). For comparison, for a system with Deterministic IAT and ST, we get $E[T] = 1ms$. Thus, the added variability due to an Exponential IAT already increases mean latency by 70%. For the $M/G/1$ system, if we now consider Exponential ST ($C^2_{ST} = 1$), we get $E[T] = 2.5ms$, a 250% increase over the baseline. If we further increase the variability in ST to $C^2_{ST} = 2$ using a H_2 distribution, we get $E[T] = 3.3ms$, a 330% increase! The increase in request latency with variability is even more pronounced at higher loads.

While the above results highlight the importance of reducing variability in ST, we note that making changes to a system to reduce variability may lead to other performance overheads, resulting in higher $E[ST]$. For example, the OS scheduler may opportunistically move threads between cores to leverage idle cores. However, this introduces ST variability due to the associated state migration and context switches. Disabling this opportunistic behavior can mitigate variability, but may lead to scenarios where threads are waiting on a busy core even though other cores are idle. There is thus a trade-off between reducing $Var(ST)$ and increasing $E[ST]$, which is captured by Eq. (3).

Figure 1 illustrates the impact on mean request latency of the trade-off between $E[ST]$ and $Var(ST)$ for the $M/G/1$ model. We set $\lambda = 100$ req/s, and use an H_2 distributed ST whose variability can be controlled. To highlight the trade-off, we show the equi-latency line of $5ms$ in black. Point B on this line has a *higher* $E[ST]$ but *lower* $Var(ST)$ compared to point A; nonetheless, both have the same $5ms$ latency. Thus, we can afford some overhead in $E[ST]$ when reducing $Var(ST)$ of a system. Finally, point C has the same $E[ST]$ as A, but has much lower $Var(ST)$; as a result, the mean request latency for C is almost 30% lower than that for A.

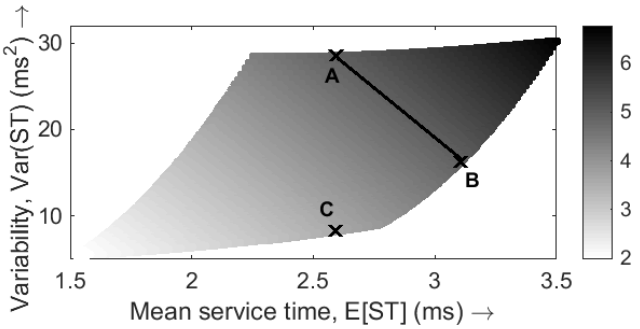

Figure 1: Heatmap of mean latency as a function of mean service time ($E[ST]$) and variability in service time ($Var(ST)$). The black diagonal line denotes the equi-latency line of $5ms$.

Key takeaways: *(i) Reducing service time variability can significantly lower request latency (by 2× or higher), and (ii) It is beneficial to reduce variability even if doing so increases the mean service time.*

Objective and scope of this work

The design of today's web server systems is primarily driven by the intent to shorten the critical path of requests, that is, reducing the mean ST. However, our analytical results above suggest an alternative solution to improving request latencies in systems – *reducing variability*. Variability is often assumed to be intrinsic to the hardware and the workload, and thus not easily controllable. However, this is not always true. While variability in IAT depends on customer request behavior, variability in ST can be regulated to some extent by modifying the application and software stack. The objective of our work is to demonstrate that this alternative viewpoint of "focusing on ST variability for designing web servers" can reveal viable solutions to reduce latency. Note, however, that we are *not* arguing against solutions that focus on reducing ST.

Solution Overview

Given a web application, our goal is to improve its latency by targeting a reduction in service time (ST) variability. However, web applications can be complex, consisting of several processes and threads that work asynchronously. Further, web requests typically pass through several software layers before completing service. Thus, we must first identify the potential software layers or components that significantly contribute to ST variability, and then determine control knobs that can be tuned to reduce ST variability in the identified software layers. Some control knobs are readily available, such as OS thread scheduler, TCP congestion control, etc. However, in some software layers, effective control knobs may not be available, necessitating modifications to existing software.

We use the following methodology to achieve our goal:

(1) *Fine-grained, unobtrusive request tracing*: We employ low-overhead tracing for the web requests. Our tracing encompasses all layers of the software stack that a request goes through during its processing lifetime, including the OS and network stack.
(2) *Identifying the source(s) of service time variability*: We aggregate the tracing information across all requests using low-overhead histograms to identify the software layers with the highest variability that are amenable to modification.
(3) *Modifying the system software to mitigate variability and reduce request latency*: We explore available control knobs in the system that can reduce the observed ST variability in the target software layer, even if this reduction comes at the expense of an increase in mean ST (see Section 3.1). Once the knob is determined, we investigate its optimal setting to minimize end-to-end request latency.

Request Tracing

Our request tracing works by timestamping the request through several layers starting from when it arrives at the host from the server's NIC until immediately before the application transfers the response packet back to the OS. Given our focus on variability in service times, we only trace the request at the host server, and not the client.

To store the timestamps, we append an empty 64-byte buffer to the original request packet, similar to prior works [15]. Then, as the request goes through different stages of processing on the client and the server, timestamps are recorded at appropriate offsets for the stage of processing by writing the system clock time into the buffer; we use the system clock with nanosecond precision to record timestamps within the server. By appending the small buffer to the request, we can record multiple timestamps and track the request to which they correspond without requiring any additional post-processing. The above tracing implementation required modification to the Linux kernel source, network drivers, and the application protocols to write timestamps into the appended buffer at the right offset. The overhead on request latency through all layers is low, about 5%; this overhead is incurred by both the baseline and our approach.

To choose the timestamping locations, we consider all possible components within the host server that may contribute to service time variability; these are locations where significant request processing may occur. For this study, we timestamp at the following locations/events at the host server:

e1: In the host network driver when the packet arrives at the NIC.
e2: At the end of TCP processing.
e3: When the application drains the requests from the socket.
e4: When the application starts processing an individual request.
e5: When the application hands off the response to the kernel.
e6: When the response is dispatched from the host server's NIC.

Timestamping at these different event boundaries provides us an opportunity to analyze the time spent by the requests at different stages, T_{ei}, for $i = 1, 2, \ldots 6$, as shown in Figure 2. However, other fine grained events can be used when required.

- **driver-to-tcp**: $(T_{e2} - T_{e1})$ is the time spent by the request from driver to TCP layer, representing the network stack processing delay.
- **tcp-to-socket**: $(T_{e3} - T_{e2})$ is the time spent between the TCP layers and the socket, and represents the wakeup/scheduling delay.

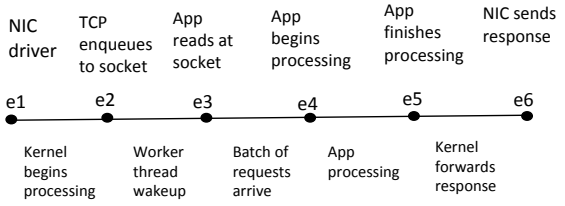

Figure 2: Timestamping locations in our request tracing.

- **socket-to-parse**: $(T_{e4} - T_{e3})$ is the time between application processing and socket, and represents the queuing delay at the application level caused by batching of requests.
- **parse-to-response**: $(T_{e5} - T_{e4})$ is the user-space application processing time.
- **response-to-send**: $(T_{e6} - T_{e5})$ is the network stack processing time to dispatch the response from the host server.

To efficiently compute variability at each stage, we maintain a running sum (across requests) of X and X^2, where X is the time spent in a stage. At the end of an experiment, we compute the sample moments $E[X]$ and $E[X^2]$ by dividing the running sums by the number of requests. Finally, we compute $Var(X) = E[X^2] - (E[X])^2$.

In our evaluation, we find that the most significant stages, in terms of variability, are the tcp-to-socket, socket-to-parse, and parse-to-response. We will discuss their role in request processing in detail in the evaluation sections. We also traced the request on the outgoing send path (server socket to driver), but the variability on this path is much smaller, only about 20% of the variability on the incoming path from client to server. Unless otherwise noted, we report $(T_{e6} - T_{e1})$ as the latency of a given request. This definition only includes the server-side latency, which is the focus of our current work.

Evaluation

We explore the methodology presented in Sections 4 and 5 in the following use cases:

(1) *Request batching at the Memcached client at high throughput*: Using our profiling approach, we find that the *socket-to-parse* stage of processing exhibits the highest service time variability. To reduce the network overhead of sending short packets, the Linux kernel at the client batches incoming requests until it gets a response back from the server saying it has received the previous batch (Nagle's algorithm [28]). However, since network conditions can be variable, this default batching behavior leads to bursts of processing at the server, followed by idle times, resulting in significant service time variability. We modify this default batching behavior to closely regulate the time between batches, lowering the variability (by as much as 76%) and improving tail latency by up to 40%.

(2) *Redesigning the LRU management on the Memcached server at low throughput*: In the case of low throughput configuration, we find that the *tcp-to-socket* stage of processing exhibits the highest service time variability. The Memcached server reorders items to maintain the Least-Recently-Used (LRU) ordering of items in the cache. To shorten the critical path, Memcached moves the LRU maintenance off the request processing path and delegates the responsibility to a dedicated LRU management thread that is run periodically. When this thread is active, it interferes with the Memcached request processing, resulting in high service time variability. We redesign the LRU management functionality and include it on the critical path of Memcached request processing in the form of fine-grained slices of LRU work. This counter-intuitive redesign reduces variability and improves latency by about 30%.

(3) *Application thread pinning for the Apache web server*: Using our profiling approach, we find that the *socket-to-parse* stage of processing exhibits the highest service time variability at the Apache web server. The Apache web server schedules its various worker threads and processes on any available idle CPU core opportunistically to start serving customer requests as soon as possible. Consequently, threads may move between cores, resulting in context switch overheads and state migration. We explore thread pinning to reduce this overhead and associated variability, though at the expense of possibly delaying request processing when the pinned core is busy. We show that, at high load, this approach can reduce latency by 50%.

We experimentally evaluate the performance improvements for the above three use cases under various workload scenarios, including different levels of load, different inter-arrival time distributions, and different (time-varying) arrival traces. Our alternative approach of focusing on service time variability substantially reduces mean and tail latency (by up to 30–50%) in all three cases. Importantly, we do so by simply changing the metric that we use to identify the critical stage of request processing and determine the appropriate control knob.

To highlight the significance of using variability as a guiding principle, we also evaluate the use of existing approaches, such as exclusively focusing on reducing mean service time or employing ad-hoc control knobs, and show that such approaches can end up *hurting* request latency. For example, when employing mean service time as the metric to determine the bottleneck stage of request processing for the Memcached application in the low throughput configuration, the socket-to-parse stage comes out on top (instead of the tcp-to-socket stage determined by our approach of using variability as the metric). Accordingly, if we employ batching to mitigate the time spent in the socket-to-parse stage, we find that latency does improve at low request rates, but latency suffers, by as much as 32%, at high request rates. By contrast, our approach consistently improves latency by about 26-61% when employing the amortized LRU to mitigate variability in the tcp-to-socket stage. This shows that exclusively focusing on reducing mean service time may not always improve latency; a more robust approach is to also consider service time variability.

Selecting control knobs in an ad-hoc manner is an alternative and sometimes easier approach. Consider the Memcached application; an alternative control knob is to pin application threads. However, under the high throughput configuration, by employing thread pinning, we find that mean latency increases by about 11.5%, whereas our chosen control knob (batching) improves latency by

about 20.9%. Likewise, for the low throughput configuration, pinning threads hurts latency by about 51.1%, whereas our chosen control knob (amortized LRU) improves latency by about 35.4%. Similarly, for the Apache web server, another option is to batch requests by modifying the Nagle's algorithm. However, even under the optimal batching interval for Apache, the improvement in mean request latency is only about 2.7%, whereas that under our chosen control knob (pinning) is about 21.1%. We refer interested readers to our full evaluation results for further details [20].

Conclusion

Latency is a critical metric for user-facing web services. Existing work typically focuses on shortening the critical path (or mean service time) to improve performance. This tutorial takes an alternative approach to improving application performance - reducing the variability in request processing. By using "reduction in service time variability" as the guiding principle in web server design, we reveal control knobs that improve request latency (both mean and tail) by up to 28-50% across different scenarios. Our end-goal is to make the case for using service time variability as a metric to guide system design. The three use cases presented in this tutorial demonstrate the validity of this approach.

As part of future work, we plan to extend our approach to mitigate variability in multi-tier and microservices architectures. In such cases, it is common for a bottleneck in one stage/service to have a cascading effect on other components of the application [23]. Further, the latency in such architectures is known to exhibit high variability [6], suggesting that our approach can be useful in such settings.

Biographical Sketches

Anshul Gandhi is an Assistant Professor in the Computer Science Department at Stony Brook University. He received his Ph.D. from Carnegie Mellon University in 2013 and then spent a year as a postdoc at the IBM T. J. Watson Research Center. His current research focuses on performance modeling in distributed systems, and is funded by an NSF Career award, an IBM Faculty award, and a Google Research award. His contributions to performance modeling were recently recognized by an ACM Sigmetrics Rising Star Award.

Amoghavarsha Suresh is a Ph.D. student in the Computer Science Department at Stony Brook University advised by Anshul Gandhi. His research interests are in performance modeling of systems and computer architecture. He has published his research in various leading venues, including MICRO, WWW, and SOCC.

Acknowledgment

This work was supported by NSF CNS grants 1617046, 1717588, and 1750109.

References

[1] Atikoglu et al. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. London, England, UK, 53–64.

[2] Brin et al. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the 7th International World-Wide Web Conference*.

[3] DeCandia et al. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. Stevenson, Washington, USA, 205–220.

[4] Dean et al. 2013. The Tail at Scale. *Communications of ACM* 56, 2 (2013), 74–80.

[5] Gill et al. 2007. Youtube Traffic Characterization: A View from the Edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC '07)*. San Diego, CA, USA, 15–28.

[6] Gan et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[7] Hoffmann et al. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. Newport Beach, CA, USA, 199–212.

[8] Haque et al. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. *SIGPLAN Not.* 50, 4 (March 2015), 161–175. https://doi.org/10.1145/2775054.2694384

[9] Harchol-Balter et al. 2003. Size-based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems* 21, 2 (2003), 207–233.

[10] Jalaparti et al. 2013. Speeding Up Distributed Request-response Workflows. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 219–230. https://doi.org/10.1145/2534169.2486028

[11] Kamra et al. 2004. Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites. In *Proceedings of the Twelfth IEEE International Workshop on Quality of Service*. Montreal, Canada, 47–56.

[12] Kapoor et al. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. San Jose, CA, USA, Article 9, 9:1–9:14 pages.

[13] Kanev et al. 2015. Profiling a Warehouse-scale Computer. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[14] Liu et al. 2011. Greening Geographical Load Balancing. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. San Jose, CA, USA, 233–244.

[15] Li et al. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2670979.2670988

[16] Meisner et al. 2009. PowerNap: eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. Washington, DC, USA, 205–216.

[17] Meisner et al. 2011. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. San Jose, CA, USA, 319–330.

[18] Reiss et al. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*. San Jose, CA, USA, Article 7.

[19] Schurman et al. 2009. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. http://velocityconf.com/velocity2009/public/schedule/detail/8523.

[20] Suresh et al. 2019. Using Variability As a Guiding Principle to Reduce Latency in Web Applications via OS Profiling. In *The World Wide Web Conference (WWW '19)*. ACM, New York, NY, USA, 1759–1770. https://doi.org/10.1145/3308558.3313406

[21] Urgaonkar et al. 2005. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*. Banff, Alberta, Canada, 291–302.

[22] Urgaonkar et al. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC '05)*. Seattle, WA, USA, 217–228.

[23] Wang et al. 2017. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 207–217. https://doi.org/10.1109/ICDCS.2017.32

[24] Zhang et al. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*. Taipei, Taiwan, 743–752.

[25] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (2004).

[26] Tabb Group. 2008. The Value of a Millisecond: Finding the Optimal Speed of a Trading Infrastructure. http://www.tabbgroup.com/PublicationDetail.aspx?PublicationID=346.

[27] Leonard Kleinrock. 1975. *Queueing Systems, Volume I: Theory*. Wiley-Interscience.

[28] John Nagle. 1984. Congestion Control in IP/TCP Internetworks. *SIGCOMM Computer Communication Review* 14, 4 (1984), 11–17.

[29] The Apache Software Foundation. 2019 (Accessed: 1/21/2019). Apache HTTP Server Project. https://httpd.apache.org.