Seer: Leveraging Big Data to Navigate The Complexity of Cloud Debugging

Yu Gan Meghna Pancholi Dailun Cheng Siyuan Hu Yuan He Christina Delimitrou
Cornell University
{yg397,mp832,dc924,sh2442,yh772,delimitrou}@cornell.edu

Abstract

Performance unpredictability in cloud services leads to poor user experience, degraded availability, and has revenue ramifications. Detecting performance degradation a posteriori helps the system take corrective action, but does not avoid the QoS violations. Detecting QoS violations after the fact is even more detrimental when a service consists of hundreds of thousands of loosely-coupled microservices, since performance hiccups can quickly propagate across the dependency graph of microservices. In this work we focus on anticipating QoS violations in cloud settings to mitigate performance unpredictability to begin with. We propose *Seer*, a cloud runtime that leverages the massive amount of tracing data cloud systems collect over time and a set of practical learning techniques to signal upcoming QoS violations, as well as identify the microservice(s) causing them. Once an imminent QoS violation is detected Seer uses machine-level hardware events to determine the cause of the QoS violation, and adjusts the resource allocations to prevent it. In local clusters with 10 40-core servers and 200-instance clusters on GCE running diverse cloud microservices, we show that Seer correctly anticipates QoS violations 91% of the time, and attributes the violation to the correct microservice in 89% of cases. Finally, Seer detects QoS violations early enough for a corrective action to almost always be applied successfully.

1 Introduction

Cloud computing services are governed by strict quality of service (QoS) constraints in terms of throughput, and more critically tail latency [8, 12, 14, 15]. Violating these requirements worsens the end user experience, leads to loss of availability and reliability, and has severe revenue implications [7, 8, 12, 17, 18]. A recent shift from monolithic designs to loosely-coupled microservices is aimed at improving service deployment, isolation, and modularity, but at the same time puts more pressure on performance predictability, as the latency requirements of each individual microservice is often in the microsecond granularity. Similarly, as datacenter servers become increasingly heterogeneous with the addition of FPGAs, hardware accelerators, and network offload engines, performance predictability becomes even more challenging.

The need for performance predictability has prompted a long line of work on performance tracing, monitoring, and debugging systems [11, 22, 28, 29, 32, 33]. Systems like Dapper and GWP rely on distributed tracing (often at RPC level) to detect performance abnormalities, while the Mystery Machine [11] leverages the large amount of logged data to extract the causal relationships between messages, and sidestep the challenge of clock synchronization across large clusters. This dependency model between requests can then be used towards performance optimizations, such as incremental result propagation that leverages the latency slack of certain requests.

In this work we take performance debugging one step further. Specifically, detecting QoS violations after the fact, although useful to amend prolonged degraded performance caused by events like misconfigurations and machine failures, still incurs the poor user experience and revenue implications discussed above. Even more, the longer the system operates oversubscribed, the longer it takes for corrective actions to take effect and restore QoS. This is especially true when applications consist of microservices where backpressure effects between application tiers can cause bottlenecks to propagate and amplify through the system. In microservices-based applications, performance debugging also has the additional challenge of pinpointing the culprit of a QoS violation, a non-trivial task given the complexity of dependencies between microservices in production systems.

Given the consequences of QoS violations, we set out to answer the following questions: *(i) can QoS violations be anticipated in cloud systems that host microservices-based applications, and (ii) can we pinpoint which microservice is the culprit of an upcoming QoS violation early enough to take corrective action?*

Initially, anticipating performance degradations seems infeasible given that the vast majority of QoS violations are caused by unpredictable, short-term transient effects [30]. An aid in this attempt is the massive amount of data cloud systems collect about the execution of services they host over time. By mining this information in a practical, online manner we can detect QoS violations just early enough to avoid them altogether by taking actions, such as adjusting resource allocations.

We present *Seer*, a cloud monitoring and performance debugging system that leverages deep learning to diag-

nose and prevent QoS violations in a practical and online manner. The NN in Seer is trained offline on annotated, RPC-level execution traces collected using Apache Thrift's timing interface [1]. At runtime, Seer takes streaming traces as input, and outputs the microservice (if any) that will cause a QoS violation in the near future. Traces capture the queue depth in front of each microservice at fine-grained intervals; we also experimented with latency and utilization traces and show that unlike queue depths, they do not correlate closely with performance. While inference is fast for small clusters, as systems scale inference time does to. To speedup performance debugging at scale, we offloaded inference on Google's TPU cloud, which accelerated inference by almost two orders of magnitude. The current design converges within 5-10ms for a neural network with several hundred input and output neurons, and 5 hidden layers.

We evaluate Seer both in our local cluster of 10 40-core machines, and on a 200-instance cluster on Google Compute Engine. In our local cluster Seer correctly identifies upcoming QoS violations in 93% of cases, and correctly pinpoints the microservice initiating the QoS violation 89% of the time. In the GCE cluster, it correctly detects QoS violations 90% of the time, and correctly identifies the culprit in 86% of cases. For the cases where QoS violations are anticipated correctly, Seer is able to adjust resource allocations to prevent them altogether in most cases. As cloud systems become increasingly complex, systems like Seer that take a data-driven approach can make their management more practical. We are currently working to make the system more scalable and robust to server heterogeneity, missing or noisy input traces, and techniques like autoscaling.

2 The Design of Seer

2.1 Distributed Tracing

A major challenge with microservices is that one cannot simply rely on the client to report performance as with traditional client-server applications. We developed a distributed tracing system that records latencies at RPC granularity using the Thrift timing interface. RPCs and REST requests are timestamped upon arrival and departure from each microservice by the tracing module, and data is aggregated in a centralized Cassandra database. The design of the tracing system is similar to Zipkin [6]. We additionally track the number of requests queued in each microservice, and distinguish between the time spent processing network requests and the time that goes towards application computation. In all cases the overhead from tracing is negligible, less than 0.1% on end-to-end latency, which is tolerable for such systems [11, 28, 29].

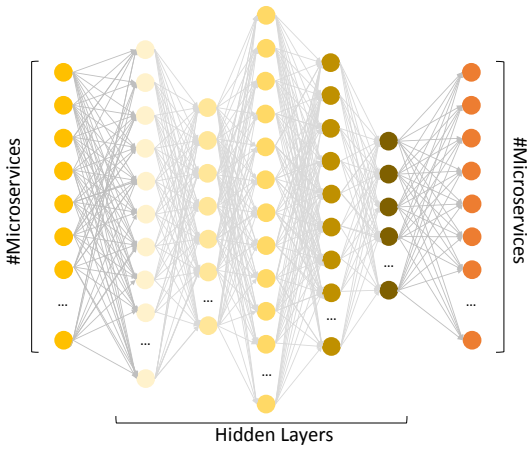

Figure 1: The neural network design in Seer.

2.2 Learning in Performance Debugging

A popular way to model performance in cloud systems, especially when there are dependencies between tasks, are *queueing networks*. Although queueing networks are a valuable tool to model how bottlenecks propagate through the system, they require in-depth knowledge of application structure, and can become overly complex as applications and systems scale. They additionally cannot easily capture all sources of contention such as the operating system and network stack.

Instead in Seer, we take a data-driven approach that assumes no a priori knowledge on the architecture of a service, making the system robust to changing and unknown applications. The key idea is that conditions that led to QoS violations in the past can be used to anticipate QoS violations in the near future. A deep neural network is trained on the distributed traces collected using the monitoring system above to anticipate future QoS violations. There are two main factors that impact the network's accuracy; the metric that is used as input, and the configuration of the network's neurons and layers. We experimented with resource utilization, latency, and queue depths as input metrics. Consistent with prior work, utilization was not a good proxy for performance [14, 17, 25, 26]. Similarly latency led to a large number of false positives, or incorrectly signaled computationally-intensive microservices as the QoS violation culprits. Again consistent with queueing theory [24] and prior work [16, 19, 23, 25], per-microservice queue depths consistently captured performance bottlenecks and pinpointed the microservices causing them.

The second challenge, tuning the configuration parameters in the network (learning rate a, hidden layers, batch size, hidden units per layer) is done empirically. Figure 1 shows the neural network in Seer. The number of input and output neurons is equal to the number of active microservices in the cluster, with each input neuron captur-

Service	Protocol	Per-language LoC breakdown of end-to-end service
Social network	**RPC**	34% C, 23% C++, 18% Java, 7% node.js, 6% Python, 5% Scala, 3% PHP, 2% Javascript, 2% Go
Movie streaming	**RPC**	30% C, 21% C++, 20% Java, 10% PHP, 8% Scala, 5% node.js, 3% Python, 3% Javascript
E-commerce	**REST**	21% Java, 16% C++, 15% C, 14% Go, 10% Javascript, 7% node, 5% C#, 5% Scala, 4% HTML, 3% Ruby

Table 1: Code composition of each end-to-end service.

ing the queue depth of the corresponding microservice, and each output neuron firing if/when that microservice is about to initiate a QoS violation in the near future. All microservices in our setting run in single-concerned Docker containers, i.e., only a single microservice runs per container. This simplifies scaling up/out individual microservices independently. In Section 5 we discuss the implications of the number of active microservices changing as a result of techniques like autoscaling. The learning rate a is configured using `ADAGRAD` [20], keeping the number of neurons constant. We then explore the impact of the number of hidden layers and units per hidden layer on output quality. The five hidden layers shown in Fig. 1 maximize the detection accuracy across a diverse set of application and system configurations, disjoint from the trace set the network is trained on (see Validation section below). Weights and biases are obtained via Stochastic Gradient Descent (SGD) [9, 31].

Training process: Seer is trained on execution traces collected from all active microservices over time. Training happens offline, and only needs to be repeated when the server configurations or the type of active microservices change substantially. Traces from multiple servers are synchronized, and include requests queued per microservice over time. Training traces include annotated QoS violations; for now annotation is supervised manually, however we are exploring ways to completely automate the annotation process.

Inference process: During normal operation, execution traces are streamed through the network every few milliseconds and potential upcoming QoS violations are signaled. Once an imminent QoS violation is detected, Seer takes action by first determining why the microservice is misbehaving, and then adjusting the resource allocation of the offending microservice to mitigate the unpredictable performance. In Section 4 we show an example of system behavior with and without Seer's intervention.

Why deep learning? Although deep learning is not the only approach that can be used for proactive QoS violation detection, there are several reasons why it is preferable in this case. First, the problem Seer must solve is a pattern matching problem of recognizing queueing patterns between microservices that result in QoS violations, where the patterns are not always known or easy to annotate. This is a more complicated task than simply signaling a microservice with many enqueued requests, for which simpler classification, regression, or sequence labeling techniques would suffice. Second, the DNN in Seer assumes no a priori knowledge about the structure and dependencies between individual microservices, making it applicable for services where the application architecture changes frequently, is overly complex for users to manually express dependencies, or for public cloud settings where the cloud provider does not have access to the application source code. Third, deep learning has been shown to be especially effective in pattern recognition problems with massive datasets, e.g., in image or text recognition. Finally, as we show in the validation section below, using deep learning allows Seer to recognize QoS violations with high accuracy, and within the opportunity window the resource manager needs to apply corrective actions.

3 Validation

3.1 Methodology

Applications: Although there are many open-source microservices that could serve as individual components of an end-to-end service, there are no representative end-to-end applications built with microservices, with the exception of Sockshop, an e-commerce site by Weave [5]. To address this we have developed three end-to-end services which we plan to open-source, each consisting of tens of different microservices, and implementing a *social network*, a *movie reviewing/streaming* service, and an *e-commence site* based on Sockshop. Individual microservices include `nginx` [4], `memcached` [21], `mongodb` [3], `RabbitMQ` [2], and `http server`, among others. Table 1 shows a breakdown of each end-to-end service per language, which highlights the software heterogeneity that is often present in microservices. We additionally built an RPC framework over Apache Thrift [1] to connect individual microservices in the *social network* and *movie streaming* service. Microservices in the *e-commerce site* are connected over http.

Systems: First, we use a dedicated local cluster with 10, 2-socket 40-core servers with 128GB of RAM each. Each server is connected to a 40Gbps ToR switch over 10Gbe NICs. Second, we use a 200-instance cluster on Google Compute Engine (GCE) to study the scalability of Seer. All instances are `n1-standard-64`, each with 64 vCPUs and 240GB of RAM.

3.2 Evaluation

Accuracy: Fig. 2a shows the detection accuracy in Seer under different input metrics. CPU utilization and per-microservice latencies miss the majority of QoS viola-

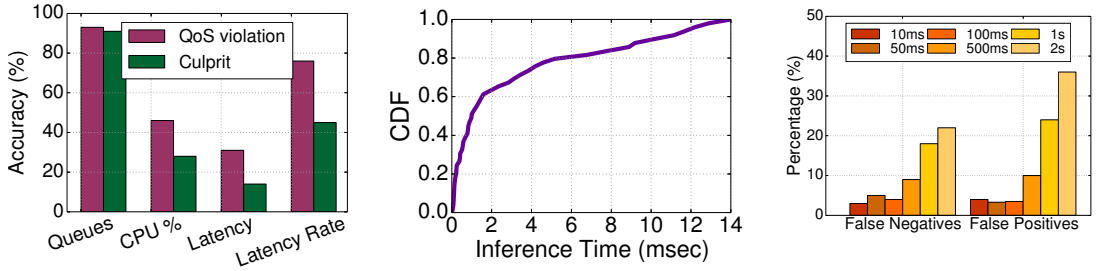

Figure 2: (a) The accuracy of detecting upcoming QoS violations when using different metrics as inputs of the neural network. (b) The time it takes for inference to converge in Seer, for the small-scale 10 server cluster. (c) The false negatives and false positives in Seer as we vary the prediction window.

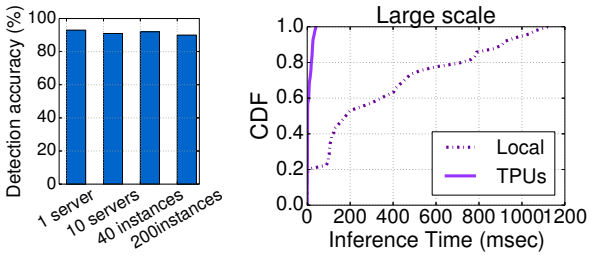

Figure 3: (a) The accuracy of QoS violation detection is Seer as the cluster size increases, and (b) The CDF of convergence time with our local implementation, and using the Google TPU Cloud for the 200-instance cluster.

tions and mislabel the microservices initiating the violations. Using the rate at which per-microservice latency changes achieves higher accuracy but still misses a significant fraction of QoS violations, and incorrectly labels a disproportionate fraction of microservices as culprits. Using the per-microservice queue depth as the input of the neural network captures the majority of QoS violations, and pinpoints the responsible microservice(s).

Inference time: Fig. 2b shows the convergence time for inference in the small 10-server cluster. For 60% of QoS violations, detection happens within 2ms from obtaining the per-microservice traces, early enough to apply most corrective actions that avoid the QoS violation altogether. Even the QoS violations in the high percentiles of the CDF are detected within 14ms at most, which is usually sufficient for the system to react.

False negatives & positives: Fig. 2c shows the percentage of false negatives and false positives in Seer as we vary the prediction window. When the prediction window is 10-100ms both false positives and false negatives are low, as Seer uses a very recent snapshot of the cluster state to anticipate performance unpredictability. If inference was instantaneous, very short prediction windows would always be better, as they reflect the current state of the cluster. However, given that inference takes several milliseconds and more importantly, applying corrective actions to avoid QoS violations takes several tens of milliseconds to take effect, such short windows defy the point of proactive QoS violation detection. At the other end, predicting far into the future results in significant false negatives, and especially false positives. This is because many QoS violations are caused by very short, bursty events that do not have an impact on queue lengths until a few milliseconds before the violation occurs. Therefore requiring Seer to predict one or more seconds into the future means that normal queue depths are annotated as QoS violations, resulting in many false positives. Unless otherwise specified we use a 100ms prediction window for the remainder of the paper.

Scalability: We now examine Seer as the cluster size increases. Fig. 3a shows QoS detection accuracy for different cluster sizes; the 1 and 10 server settings are local, while the 40- and 200-instance clusters are on GCE. Seer is robust to the size of the cluster in terms of detection accuracy, although as seen in Fig. 3b for the 200-instance cluster, the penalty of scalability comes in terms of inference time. For the majority of cases inference takes several hundreds of ms, at which point the QoS violation has occurred. To address this we rewrote Seer using Tensorflow and ported it on the Google TPU public cloud. The change in inference time is dramatic, two orders of magnitude in some cases, ensuring that detection happens early enough to be meaningful.

4 QoS Violation Prevention

Once an upcoming QoS violation is detected, Seer takes action to try to avoid it. This involves first determining what will cause the QoS violation before it manifests as an increase in tail latency. To do so Seer looks at hardware-level per-resource utilization statistics on the machine where the offending microservice resides. This includes CPU utilization, memory, network, and I/O bandwidth usage, and last level cache misses. Although this is not an exhaustive list of resources where contention can emerge, in practice it covers a large fraction of performance degradations.

Once the problematic resource is located, Seer adjusts the resource allocation, either resizing the Docker container, or using mechanisms like Intel's Cache Alloca-

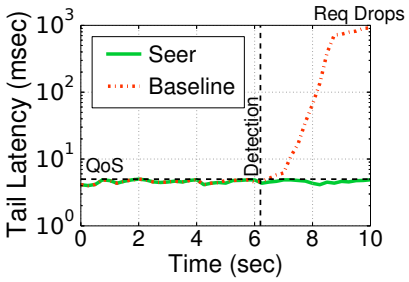

Figure 4: Tail latency with and without Seer.

tion Technology (CAT) for last level cache (LLC) partitioning, and the Linux traffic control's hierarchical token bucket (HTB) queueing discipline in `qdisc` [10, 27] for network bandwidth partitioning.

Fig. 4 shows the impact on tail latency with and without Seer. Once the upcoming QoS violation is detected Seer determines the problematic resource, in this case insufficient LLC capacity, and uses CAT to adjust it. Post detection the service's tail latency with Seer remains nominal, while if the QoS violation had remained undetected tail latency would continue to worsen until requests started getting dropped. Note that once the system arrives in such a problematic state it, returning to normal operation has significant inertia.

Some of the measurements Seer uses for detect problematic resources involve access to hardware performance counters. Unfortunately public clouds do not enable access to such events. In that case, Seer uses a set of contentious microbenchmarks, each targeting a different system resource to pinpoint problematic resources [13]. For example, a cache thrashing microbenchmark will reveal cache saturation, while a network bandwidth-demanding microbenchmark will reveal insufficient bandwidth allocations. These microbenchmarks need to run for a few 10s of milliseconds before signaling the resource under contention.

5 Discussion

Seer is currently used by several groups at Cornell and elsewhere. Nonetheless, the present design has a number of limitations, which we are currently addressing. First, because the number of input and output neurons in Seer is equal to the number of active microservices, the system needs to be retrained if techniques like autoscale which spawn additional containers, or terminate existing containers are present. The same applies when the architecture of the end-to-end application changes, e.g., when more applications are decomposed to microservices, or new features are added to the service. Currently we use a *shadow* DNN, retrained in the background to adjust to changes in the application architecture. While retraining happens, Seer uses the primary network to anticipate QoS violations, which may lead to undetected unpredictable performance. We are exploring more practical ways to make the network robust to application changes.

Second, Seer currently assumes no knowledge about the structure of the end-to-end service. We are exploring whether users expressing the application architecture, or the system learning it via the tracing system can improve the accuracy and/or scalability of Seer.

Third, Seer assumes full control over the cluster, or at least over individual servers, such that it can collect traces from all active microservices. This may not always be the case, especially in public clouds, or when using third-party applications that cannot easily be instrumented. We are extending the system design to tolerate missing or noisy tracing information.

Finally, even though Seer is able to avert the majority of QoS violations, there are still some events that are not predicted early enough for corrective actions to take place. These typically involve memory-bound microservices, where the memory subsystem is saturated. Memory, like any storage medium, has inertia, so resource adjustment decisions require more time to take effect. We are exploring whether predicting further into the future is possible without significantly increasing the number of false positives, or whether alternative resource isolation mechanisms like cache partitioning can help alleviate memory pressure faster.

6 Future Work

Cloud systems and applications continuously increase in size and complexity. The recent switch from monoliths to microservices puts even more pressure on performance predictability, and at the same time makes manual performance debugging impractical. In this paper we presented early work on Seer, a monitoring and performance debugging runtime that leverages practical learning techniques and the massive amount of tracing data cloud systems collect, to proactively detect QoS violations with enough slack to prevent them altogether. We have evaluated Seer both on local clusters and a large cluster on GCE, and validated its accuracy in anticipating QoS violations and pinpointing the microservices that cause them. As cloud and IoT applications continue to shift from batch to low-latency, systems like Seer can improve their QoS, predictability, and responsiveness in a practical and online manner.

Acknowledgements

We sincerely thank Lydia Chen, and the anonymous reviewers for their feedback on this manuscript. This work was supported by a Facebook Faculty Research Award, a John and Norma Balen Sesquicentennial Faculty Fellowship, and gifts from VMWare Research and Intel.

References

[1] Apache thrift. https://thrift.apache.org.

[2] Messaging that just works. https://www.rabbitmq.com/.

[3] mongodb. https://www.mongodb.com.

[4] Nginx. https://nginx.org/en.

[5] Sockshop: A microservices demo application. https://www.weave.works/blog/sock-shop-microservices-demo-application.

[6] Zipkin. http://zipkin.io.

[7] BARROSO, L. Warehouse-scale computing: Entering the teenage decade. *ISCA Keynote, SJ, June 2011*.

[8] BARROSO, L., AND HOELZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[9] BOTTOU, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*, Paris, France, 2010.

[10] BROWN, M. A. Traffic control howto. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[11] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 217–231.

[12] DEAN, J., AND BARROSO, L. A. The tail at scale. In *CACM, Vol. 56 No. 2*.

[13] DELIMITROU, C., AND KOZYRAKIS, C. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, Portland, OR, September 2013.

[14] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, USA, 2013.

[15] DELIMITROU, C., AND KOZYRAKIS, C. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*, May/June 2014.

[16] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of ASPLOS*, Salt Lake City, 2014.

[17] DELIMITROU, C., AND KOZYRAKIS, C. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (April 2016).

[18] DELIMITROU, C., AND KOZYRAKIS, C. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proc. of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[19] DELIMITROU, C., SANCHEZ, D., AND KOZYRAKIS, C. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)* (August 2015).

[20] DUCHI, J., HAZAN, E., AND SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. In *Journal of Machine Learning Research 12 (2011) 2121-2159*, 2011.

[21] FITZPATRICK, B. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*.

[22] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 20–20.

[23] GENG, Y., LIU, S., YIN, Z., NAIK, A., PRABHAKAR, B., ROSENBLUM, M., AND VAHDAT, A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proc. of NSDI*, 2018.

[24] GROSS, D., SHORTLE, J. F., THOMPSON, J. M., AND HARRIS, C. M. Fundamentals of queueing theory. In *Wiley Series in Probability and Statistics, Book 627*, 2011.

[25] KASTURE, H., AND SANCHEZ, D. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)* (March 2014).

[26] LO, D., CHENG, L., GOVINDARAJU, R., BARROSO, L. A., AND KOZYRAKIS, C. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, Minneapolis, MN, 2014.

[27] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015.

[28] REN, G., TUNE, E., MOSELEY, T., SHI, Y., RUS, S., AND HUNDT, R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro* (2010), 65–79.

[29] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[30] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÃŰLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in googleâĂŹs datacenter network. In *Sigcomm '15* (2015).

[31] WITTEN, I. H., FRANK, E., AND HOLMES, G. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[32] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 619–634.

[33] YU, X., JOSHI, P., XU, J., JIN, G., ZHANG, H., AND JIANG, G. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 489–502.