Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices

Yu Gan
Cornell University
yg397@cornell.edu

Yanqi Zhang
Cornell University
yz2297@cornell.edu

Kelvin Hu
Cornell University
sh2442@cornell.edu

Dailun Cheng
Cornell University
dc924@cornell.edu

Yuan He
Cornell University
yh772@cornell.edu

Meghna Pancholi
Cornell University
mp832@cornell.edu

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

Abstract

Performance unpredictability is a major roadblock towards cloud adoption, and has performance, cost, and revenue ramifications. Predictable performance is even more critical as cloud services transition from monolithic designs to microservices. Detecting QoS violations after they occur in systems with microservices results in long recovery times, as hotspots propagate and amplify across dependent services.

We present Seer, an online cloud performance debugging system that leverages deep learning and the massive amount of tracing data cloud systems collect to learn spatial and temporal patterns that translate to QoS violations. Seer combines lightweight distributed RPC-level tracing, with detailed low-level hardware monitoring to signal an upcoming QoS violation, and diagnose the source of unpredictable performance. Once an imminent QoS violation is detected, Seer notifies the cluster manager to take action to avoid performance degradation altogether. We evaluate Seer both in local clusters, and in large-scale deployments of end-to-end applications built with microservices with hundreds of users. We show that Seer correctly anticipates QoS violations 91% of the time, and avoids the QoS violation to begin with in 84% of cases. Finally, we show that Seer can identify application-level design bugs, and provide insights on how to better architect microservices to achieve predictable performance.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304004

CCS Concepts • **Computer systems organization → Cloud computing**; **Availability**; • **Computing methodologies → Neural networks**; • **Software and its engineering → Scheduling**.

Keywords cloud computing, datacenter, performance debugging, QoS, deep learning, data mining, tracing, monitoring, microservices, resource management

ACM Reference Format:
Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304004

1 Introduction

Cloud computing services are governed by strict quality of service (QoS) constraints in terms of throughput, and more critically tail latency [14, 28, 31, 34]. Violating these requirements worsens the end user experience, leads to loss of availability and reliability, and has severe revenue implications [13, 14, 28, 32, 35, 36]. In an effort to meet these performance constraints and facilitate frequent application updates, cloud services have recently undergone a major shift from complex monolithic designs, which encompass the entire functionality in a single binary, to graphs of hundreds of loosely-coupled, single-concerned microservices [9, 45]. Microservices are appealing for several reasons, including accelerating development and deployment, simplifying correctness debugging, as errors can be isolated in specific tiers, and enabling a rich software ecosystem, as each microservice is written in the language or programming framework that best suits its needs.

At the same time microservices signal a fundamental departure from the way traditional cloud applications were designed, and bring with them several system challenges.

Specifically, even though the quality-of-service (QoS) requirements of the end-to-end application are similar for microservices and monoliths, the tail latency required for each individual microservice is much stricter than for traditional cloud applications [34, 43–45, 61, 62, 64, 67, 70, 77]. This puts increased pressure on delivering predictable performance, as dependencies between microservices mean that a single misbehaving microservice can cause cascading QoS violations across the system.

Fig. 1 shows three instances of real large-scale production deployments of microservices [2, 9, 11]. The perimeter of the circle (or sphere surface) shows the different microservices, and edges show dependencies between them. We also show these dependencies for *Social Network*, one of the large-scale services used in the evaluation of this work (see Sec. 3). Unfortunately the complexity of modern cloud services means that manually determining the impact of each pair-wide dependency on end-to-end QoS, or relying on the user to provide this information is impractical.

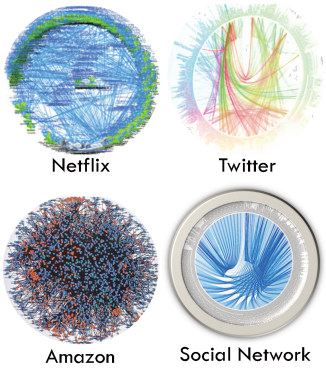

Figure 1. Microservices graphs in three large cloud providers [2, 9, 11], and our `Social Network` service.

Apart from software heterogeneity, datacenter hardware is also becoming increasingly heterogeneous as special-purpose architectures [20–22, 39, 55] and FPGAs are used to accelerate critical operations [19, 25, 40, 75]. This adds to the existing server heterogeneity in the cloud where servers are progressively replaced and upgraded over the datacenter's provisioned lifetime [31, 33, 65, 68, 95], and further complicates the effort to guarantee predictable performance.

The need for performance predictability has prompted a long line of work on performance tracing, monitoring, and debugging systems [24, 42, 46, 78, 85, 93, 97]. Systems like Dapper and GWP, for example, rely on distributed tracing (often at RPC level) and low-level hardware event monitoring respectively to detect performance abnormalities, while the Mystery Machine [24] leverages the large amount of logged data to extract the causal relationships between requests.

Even though such tracing systems help cloud providers detect QoS violations and apply corrective actions to restore performance, until those actions take effect, performance suffers. For monolithic services this primarily affects the service experiencing the QoS violation itself, and potentially services it is sharing physical resources with. With microservices, however, a posteriori QoS violation detection is more impactful, as hotspots propagate and amplify across dependent

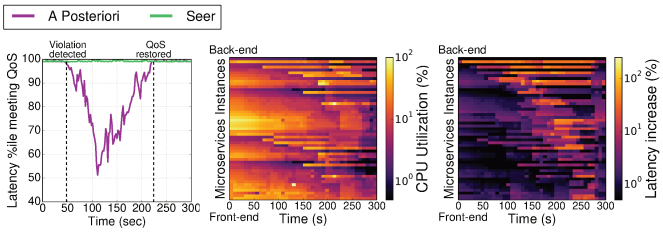

Figure 2. The performance impact of a posteriori performance diagnostics for a monolith and for microservices.

services, forcing the system to operate in a degraded state for longer, until all oversubscribed tiers have been relieved, and all accumulated queues have drained. Fig. 2a shows the impact of reacting to a QoS violation after it occurs for the *Social Network* application with several hundred users running on 20 two-socket, high-end servers. Even though the scheduler scales out all oversubscribed tiers once the violation occurs, it takes several seconds for the service to return to nominal operation. There are two reasons for this; first, by the time one tier has been upsized, its neighboring tiers have built up request backlogs, which cause them to saturate in turn. Second, utilization is not always a good proxy for tail latency and/or QoS violations [14, 28, 61, 62, 73]. Fig. 2b shows the utilization of all microservices ordered from the back-end to the front-end over time, and Fig. 2c shows their corresponding 99^{th} percentile latencies normalized to nominal operation. Although there are cases where high utilization and high latency match, the effect of hotspots propagating through the service is much more pronounced when looking at latencies, with the back-end tiers progressively saturating the service's logic and front-end microservices. In contrast, there are highly-utilized microservices that do not experience increases in their tail latency. A common way to address such QoS violations is rate limiting [86], which constrains the incoming load, until hotspots dissipate. This restores performance, but degrades the end user's experience, as a fraction of input requests is dropped.

We present *Seer*, a proactive cloud performance debugging system that leverages practical deep learning techniques to diagnose upcoming QoS violations in a scalable and online manner. First, Seer is proactive to avoid the long recovery periods of a posteriori QoS violation detection. Second, it uses the massive amount of tracing data cloud systems collect over time to learn spatial and temporal patterns that lead to QoS violations early enough to avoid them altogether. Seer includes a lightweight, distributed RPC-level tracing system, based on Apache Thrift's timing interface [1], to collect end-to-end traces of request execution, and track per-microservice outstanding requests. Seer uses these traces to train a deep neural network to recognize imminent QoS violations, and identify the microservice(s) that initiated the performance degradation. Once Seer identifies the culprit

of a QoS violation that will occur over the next few 100s of milliseconds, it uses detailed per-node hardware monitoring to determine the reason behind the degraded performance, and provide the cluster scheduler with recommendations on actions required to avoid it.

We evaluate Seer both in our local cluster of 20 two-socket servers, and on large-scale clusters on Google Compute Engine (GCE) with a set of end-to-end interactive applications built with microservices, including the *Social Network* above. In our local cluster, Seer correctly identifies upcoming QoS violations in 93% of cases, and correctly pinpoints the microservice initiating the violation 89% of the time. To combat long inference times as clusters scale, we offload the DNN training and inference to Google's Tensor Processing Units (TPUs) when running on GCE [55]. We additionally experiment with using FPGAs in Seer via Project Brainwave [25] when running on Windows Azure, and show that both types of acceleration speed up Seer by 200-235x, with the TPU helping the most during training, and vice versa for inference. Accuracy is consistent with the small cluster results.

Finally, we deploy Seer in a large-scale installation of the *Social Network* service with several hundred users, and show that it not only correctly identifies 90.6% of upcoming QoS violations and avoids 84% of them, but that detecting patterns that create hotspots helps the application's developers improve the service design, resulting in a decreasing number of QoS violations over time. As cloud application and hardware complexity continues to grow, data-driven systems like Seer can offer practical solutions for systems whose scale make empirical approaches intractable.

2 Related Work

Performance unpredictability is a well-studied problem in public clouds that stems from platform heterogeneity, resource interference, software bugs and load variation [23, 32, 34, 35, 38, 53, 58, 61–63, 72, 76, 81, 81]. We now review related work on reducing performance unpredictability in cloud systems, including through scheduling and cluster management, or through online tracing systems.

Cloud management: The prevalence of cloud computing has motivated several cluster management designs. Systems like Mesos [51], Torque [87], Tarcil [38], and Omega [82] all target the problem of resource allocation in large, multi-tenant clusters. Mesos is a two-level scheduler. It has a central coordinator that makes resource offers to application frameworks, and each framework has an individual scheduler that handles its assigned resources. Omega on the other hand, follows a shared-state approach, where multiple concurrent schedulers can view the whole cluster state, with conflicts being resolved through a transactional mechanism [82]. Tarcil leverages information on the type of resources applications need to employ a sampling-base distributed scheduler that returns high quality resources within a few milliseconds [38]. Dejavu identifies a few workload classes and reuses previous allocations for each class, to minimize reallocation overheads [90]. CloudScale [83], PRESS [48], AGILE [70] and the work by Gmach et al. [47] predict future resource needs online, often without a priori knowledge. Finally, auto-scaling systems, such as Rightscale [79], automatically scale the number of physical or virtual instances used by webserving workloads, to accommodate changes in user load.

A second line of work tries to identify resources that will allow a new, potentially-unknown application to meet its performance (throughput or tail latency) requirements [29, 31, 32, 34, 66, 68, 95]. Paragon uses classification to determine the impact of platform heterogeneity and workload interference on an unknown, incoming workload [30, 31]. It then uses this information to achieve predictable performance, and high cluster utilization. Paragon, assumes that the cluster manager has full control over all resources, which is often not the case in public clouds. Quasar extends the use of data mining in cluster management by additionally determining the appropriate amount of resources for a new application. Nathuji et al. developed a feedback-based scheme that tunes resource assignments to mitigate memory interference [69]. Yang et al. developed an online scheme that detects memory pressure and finds colocations that avoid interference on latency-sensitive workloads [95]. Similarly, DeepDive detects and manages interference between co-scheduled workloads in a VM environment [71].

Finally, CPI2 [98] throttles low-priority workloads that introduce destructive interference to important, latency-critical services, using low-level metrics of performance collected through Google-Wide Profiling (GWP). In terms of managing platform heterogeneity, Nathuji et al. [68] and Mars et al. [64] quantified its impact on conventional benchmarks and Google services, and designed schemes to predict the most appropriate servers for a workload.

Cloud tracing & diagnostics: There is extensive related work on monitoring systems that has shown that execution traces can help diagnose performance, efficiency, and even security problems in large-scale systems [12, 24, 26, 42, 54, 77, 85, 93, 97]. For example, X-Trace is a tracing framework that provides a comprehensive view of the behavior of services running on large-scale, potentially shared clusters. X-Trace supports several protocols and software systems, and has been deployed in several real-world scenarios, including DNS resolution, and a photo-hosting site [42]. The Mystery Machine, on the other hand, leverages the massive amount of monitoring data cloud systems collect to determine the causal relationship between different requests [24]. Cloudseer serves a similar purpose, building an automaton for the workflow of each task based on normal execution, and then compares against this automaton at runtime to determine if the workflow has diverged from its expected behavior [97]. Finally, there are several production systems,

Service	Communication Protocol	Unique Microservices	Per-language LoC breakdown (end-to-end service)
Social Network	**RPC**	36	34% C, 23% C++, 18% Java, 7% node, 6% Python, 5% Scala, 3% PHP, 2% JS, 2% Go
Media Service	**RPC**	38	30% C, 21% C++, 20% Java, 10% PHP, 8% Scala, 5% node, 3% Python, 3% JS
E-commerce Site	**REST**	41	21% Java, 16% C++, 15% C, 14% Go, 10% JS, 7% node, 5% Scala, 4% HTML, 3% Ruby
Banking System	**RPC**	28	29% C, 25% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python
Hotel Reservations [3]	**RPC**	15	89% Go, 7% HTML, 4% Python

Table 1. Characteristics and code composition of each end-to-end microservices-based application.

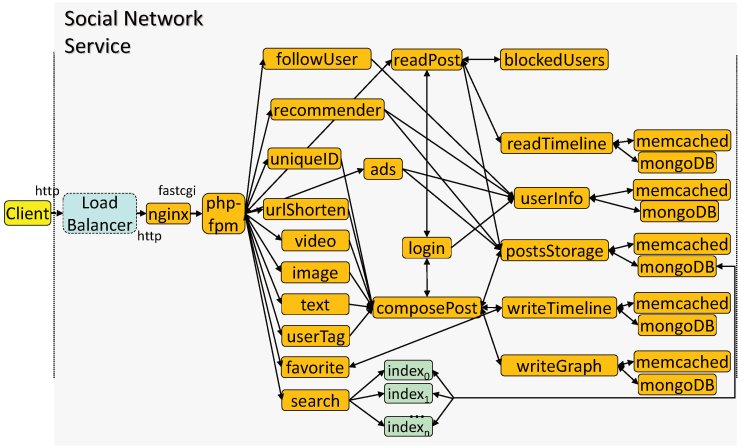

Figure 3. Dependency graph between the microservices of the end-to-end *Social Network* application.

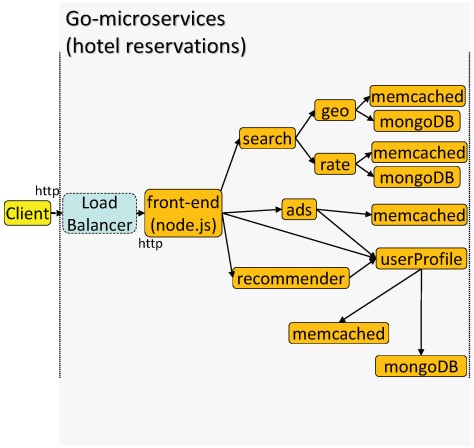

Figure 4. Architecture of the hotel reservation site using Go-microservices [3].

including Dapper [85], GWP [78], and Zipkin [8] which provide the tracing infrastructure for large-scale productions services at Google and Twitter, respectively. Dapper and Zipkin trace distributed user requests at RPC granularity, while GWP focuses on low-level hardware monitoring diagnostics.

Root cause analysis of performance abnormalities in the cloud has also gained increased attention over the past few years, as the number of interactive, latency-critical services hosted in cloud systems has increased. Jayathilaka et al. [54], for example, developed Roots, a system that automatically identifies the root cause of performance anomalies in web applications deployed in Platform-as-a-Service (PaaS) clouds. Roots tracks events within the PaaS cloud using a combination of metadata injection and platform-level instrumentation. Weng et al. [91] similarly explore the cloud provider's ability to diagnose the root cause of performance abnormalities in multi-tier applications. Finally, Ouyang et al. [74] focus on the root cause analysis of straggler tasks in distributed programming frameworks, like MapReduce and Spark.

Even though this work does not specifically target interactive, latency-critical microservices, or applications of similar granularity, such examples provide promising evidence that data-driven performance diagnostics can improve a large-scale system's ability to identify performance anomalies, and address them to meet its performance guarantees.

3 End-to-End Applications with Microservices

We motivate and evaluate Seer with a set of new end-to-end, interactive services built with microservices. Even though there are open-source microservices that can serve as components of a larger application, such as `nginx` [6], `memcached` [41], `MongoDB` [5], `Xapian` [57], and `RabbitMQ` [4], there are currently no publicly-available end-to-end microservices applications, with the exception of a few simple architectures, like Go-microservices [3], and Sockshop [7]. We design four end-to-end services implementing a *Social Network*, a *Media Service*, an *E-commerce Site*, and a *Banking System*. Starting from the Go-microservices architecture [3], we also develop an end-to-end *Hotel Reservation* system. Services are designed to be representative of frameworks used in production systems, modular, and easily reconfigurable. The end-to-end applications and tracing infrastructure are described in more detail and open-sourced in [45].

Table 1 briefly shows the characteristics of each end-to-end application, including its communication protocol, the number of unique microservices it includes, and its breakdown by programming language and framework. Unless otherwise noted, all microservices are deployed in Docker containers. Below, we briefly describe the scope and functionality of each service.

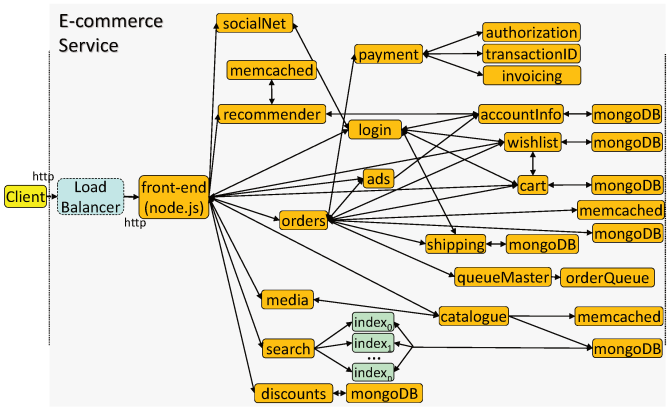

Figure 5. The architecture of the end-to-end *E-commerce* application implementing an online clothing store.

3.1 Social Network

Scope: The end-to-end service implements a broadcast-style social network with uni-directional follow relationships.

Functionality: Fig. 3 shows the architecture of the end-to-end service. Users (`client`) send requests over `http`, which first reach a load balancer, implemented with `nginx`, which selects a specific webserver is selected, also in `nginx`. Users can create posts embedded with text, media, links, and tags to other users, which are then broadcasted to all their followers. Users can also read, favorite, and repost posts, as well as reply publicly, or send a direct message to another user. The application also includes machine learning plugins, such as ads and user recommender engines [15, 16, 59, 92], a search service using `Xapian` [57], and microservices that allow users to follow, unfollow, or block other accounts. Inter-microservice messages use Apache Thrift RPCs [1]. The service's backend uses `memcached` for caching, and `MongoDB` for persistently storing posts, user profiles, media, and user recommendations. This service is broadly deployed at Cornell and elsewhere, and currently has several hundred users. We use this installation to test the effectiveness and scalability of Seer in Section 6.

3.2 Media Service

Scope: The application implements an end-to-end service for browsing movie information, as well as reviewing, rating, renting, and streaming movies [9, 11].

Functionality: As with the social network, a client request hits the load balancer which distributes requests among multiple `nginx` webservers. The front-end is similar to *Social Network*, and users can search and browse information about movies, including the plot, photos, videos, and review information, as well as insert a review for a specific movie by logging in to their account. Users can also select to rent a movie, which involves a payment authentication module to verify the user has enough funds, and a video streaming module using `nginx-hls`, a production `nginx` module for HTTP live streaming. Movie files are stored in NFS, to avoid the latency and complexity of accessing chunked records from non-relational databases, while reviews are held in `memcached` and `MongoDB` instances. Movie information is maintained in a sharded and replicated MySQL DB. We are similarly deploying *Media Service* as a hosting site for project demos at Cornell, which students can browse and review.

3.3 E-Commerce Service

Scope: The service implements an e-commerce site for clothing. The design draws inspiration, and uses several components of the open-source Sockshop application [7].

Functionality: The application front-end here is a `node.js` service. Clients can use the service to browse the inventory using `catalogue`, a Go microservice that mines the back-end `memcached` and `MongoDB` instances holding information about products. Users can also place `orders` (Go) by adding items to their `cart` (Java). After they `log in` (Go) to their account, they can select `shipping` options (Java), process their `payment` (Go), and obtain an `invoice` (Java) for their order. Orders are serialized and committed using `QueueMaster` (Go). Finally, the service includes a `recommender` engine (C++), and microservices for creating `wishlists` (Java).

3.4 Banking System

Scope: The service implements a secure banking system, supporting payments, loans, and credit card management.

Functionality: Users interface with a `node.js` front-end, similar to *E-commerce*, to login to their account, search information about the bank, or contact a representative. Once logged in, a user can process a payment from their account, pay their credit card or request a new one, request a loan, and obtain information about wealth management options. Most microservices are written in Java and Javascript. The back-end databases are `memcached` and `MongoDB` instances. The service also has a relational database (`BankInfoDB`) that includes information about the bank, its services, and representatives.

3.5 Hotel Reservation Site

Scope: The service is an online hotel reservation site for browsing information about hotels, and making reservations.

Functionality: The service is based on the Go-microservices open-source project [3], augmented with backend databases, and machine learning widgets for advertisement and hotel recommendations. A client request is first directed to one of the front-end webservers in node.js by a load balancer. The front-end then interfaces with the search engine, which allows users to explore hotel availability in a given region, and place a reservation. The service back-end consists of `memcached` and `MongoDB` instances.

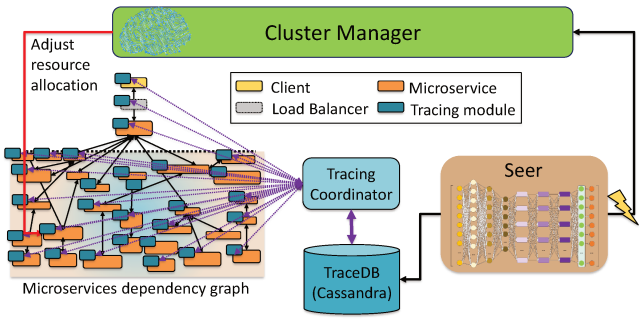

Figure 6. Overview of *Seer*'s operation.

4 Seer Design

4.1 Overview

Fig. 6 shows the high-level architecture of the system. Seer is an online performance debugging system for cloud systems hosting interactive, latency-critical services. Even though we are focusing our analysis on microservices, where the impact of QoS violations is more severe, Seer is also applicable to general cloud services, and traditional multi-tier or Service-Oriented Architecture (SOA) workloads. Seer uses two levels of tracing, shown in Fig. 7.

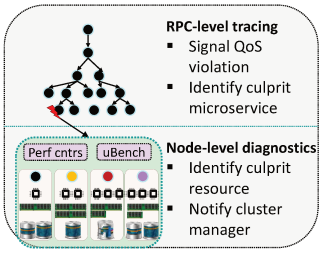

Figure 7. The two levels of tracing used in Seer.

First, it uses a lightweight, distributed RPC-level tracing system, described in Sec. 4.2, which collects end-to-end execution traces for each user request, including per-tier latency and outstanding requests, associates RPCs belonging to the same end-to-end request, and aggregates them to a centralized Cassandra database (TraceDB). From there traces are used to train Seer to recognize patterns in space (between microservices) and time that lead to QoS violations. At runtime, Seer consumes real-time streaming traces to infer whether there is an imminent QoS violation.

When a QoS violation is expected to occur and a culprit microservice has been located, Seer uses its lower tracing level, which consists of detailed per-node, low-level hardware monitoring primitives, such as performance counters, to identify the reason behind the QoS violation. It also uses this information to provide the cluster manager with recommendations on how to avoid the performance degradation altogether. When Seer runs on a public cloud where performance counters are disabled, it uses a set of tunable microbenchmarks to determine the source of unpredictable performance (see Sec. 4.4). Using two specialized tracing levels instead of collecting detailed low-level traces for all active microservices ensures that the distributed tracing is lightweight enough to track all active requests and services in the system, and that detailed low-level hardware tracing is only used on-demand, for microservices likely to cause performance disruptions. In the following sections we describe the design of the tracing system, the learning techniques in Seer and its low-level diagnostic framework, and the system insights we can draw from Seer's decisions to improve cloud application design.

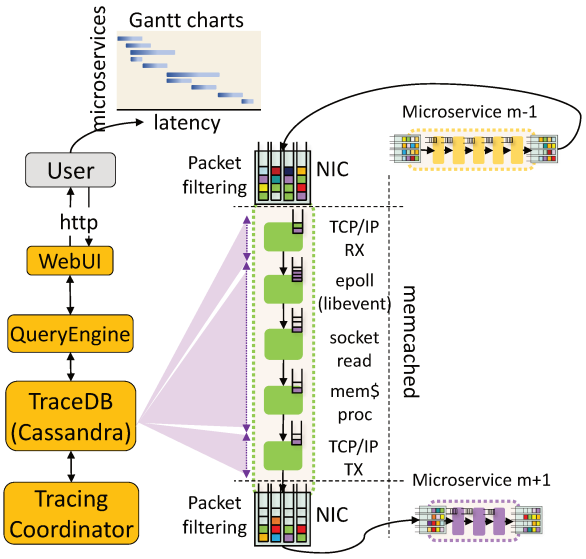

Figure 8. Distributed tracing and instrumentation in Seer.

4.2 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 have developed a distributed tracing system for Seer, similar in design to Dapper [85] and Zipkin [8] that records per-microservice latencies, using the Thrift timing interface, as shown in Fig. 8. We additionally track the number of requests queued in each microservice (*outstanding requests*), since queue lengths are highly correlated with performance and QoS violations [46, 49, 56, 96]. In all cases, the overhead from tracing without request sampling is negligible, less than 0.1% on end-to-end latency, and less than 0.15% on throughput (QPS), which is tolerable for such systems [24, 78, 85]. Traces from all microservices are aggregated in a centralized database [18].

Instrumentation: The tracing system requires two types of application instrumentation. First, to distinguish between the time spent processing network requests, and the time that goes towards application computation, we instrument the application to report the time it sees a new request (post-RPC processing). We similarly instrument the transmit side of the application. Second, systems have multiple sources of queueing in both hardware and software. To obtain accurate measurements of queue lengths per microservice, we need to account for these different queues. Fig. 8 shows an example of

Name	LoC	Layers				Nonlinear Function	Weights	Batch Size
		FC	Conv	Vect	Total			
CNN	1456		8		8	ReLU	30K	4
LSTM	944	12		6	18	sigmoid,tanh	52K	32
Seer	2882	10	7	5	22	ReLU sigmoid,tanh	80K	32

Table 2. The different neural network configurations we explored for Seer.

Seer's instrumentation for memcached. Memcached includes five main stages [60], TCP/IP receive, `epoll/libevent`, reading the request from the socket, processing the request, and responding over TCP/IP, either with the `<k,v>` pair for a read, or with an `ack` for a write. Each of these stages includes a hardware (NIC) or software (`epoll`,`socket read`,`memcached proc`) queue. For the NIC queues, Seer filters packets based on the destination microservice, but accounts for the aggregate queue length if hardware queues are shared, since that will impact how fast a microservice's packets get processed. For the software queues, Seer inserts probes in the application to read the number of queued requests in each case.

Limited instrumentation: As seen above, accounting for all sources of queueing in a complex system requires non-trivial instrumentation. This can become cumbersome if users leverage third-party applications in their services, or in the case of public cloud providers which do not have access to the source code of externally-submitted applications for instrumentation. In these cases Seer relies on the requests queued exclusively in the NIC to signal upcoming QoS violations. In Section 5 we compare the accuracy of the full versus limited instrumentation, and see that using network queue depths alone is enough to signal a large fraction of QoS violations, although smaller than when the full instrumentation is available. Exclusively polling NIC queues identifies hotspots caused by routing, incast, failures, and resource saturation, but misses QoS violations that are caused by performance and efficiency bugs in the application implementation, such as blocking behavior between microservices. Signaling such bugs helps developers better understand the microservices model, and results in better application design.

Inferring queue lengths: Additionally, there has been recent work on using deep learning to reverse engineer the number of queued requests in switches across a large network topology [46], when tracing information is incomplete. Such techniques are also applicable and beneficial for Seer when the default level of instrumentation is not available.

4.3 Deep Learning in Performance Debugging

A popular way to model performance in cloud systems, especially when there are dependencies between tasks, are *queueing networks* [49]. Although queueing networks are a valuable tool to model how bottlenecks propagate through the system, they require in-depth knowledge of application semantics and structure, and can become overly complex as applications and systems scale. They additionally cannot easily capture all sources of contention, such as the OS and network stack.

Instead in Seer, we take a data-driven, application-agnostic approach that assumes no information about the structure and bottlenecks of a service, making it robust to unknown and changing applications, and relying instead on practical learning techniques to infer patterns that lead to QoS violations. This includes both *spatial* patterns, such as dependencies between microservices, and *temporal* patterns, such as input load, and resource contention. The key idea in Seer is that conditions that led to QoS violations in the past can be used to anticipate unpredictable performance in the near future. Seer uses execution traces annotated with QoS violations and collected over time to train a deep neural network to signal upcoming QoS violations. Below we describe the structure of the neural network, why deep learning is well-suited for this problem, and how Seer adapts to changes in application structure online.

Using 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 conditions that result in QoS violations, where the patterns are not always known in advance 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 [15, 16, 92]. Second, the DNN in Seer assumes no a priori knowledge about dependencies between individual microservices, making it applicable to frequently-updated services, where describing changes is cumbersome or even difficult for the user to know. Third, deep learning has been shown to be especially effective in pattern recognition problems with massive datasets, e.g., in image or text recognition [10]. Finally, as we show in the validation section (Sec. 5), deep learning allows Seer to recognize QoS violations with high accuracy in practice, and within the opportunity window the cluster manager has to apply corrective actions.

Configuring the DNN: The input used in the network is essential for its accuracy. We have experimented with resource utilization, latency, and queue depths as input metrics. Consistent with prior work, utilization is not a good proxy for performance [31, 35, 56, 61]. Latency similarly leads to many false positives, or to incorrectly pinpointing computationally-intensive microservices as QoS violation culprits. Again consistent with queueing theory [49] and prior work [34, 37, 38, 46, 56], per-microservice queue depths accurately capture performance bottlenecks and pinpoint the

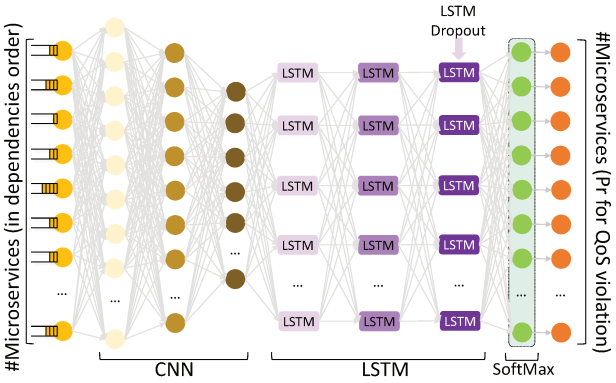

Figure 9. The deep neural network design in Seer, consisting of a set of convolution layers followed by a set of long-short term memory layers. Each input and output neuron corresponds to a microservice, ordered in topological order, from back-end microservices in the top, to front-end microservices in the bottom.

microservices causing them. We compare against utilization-based approaches in Section 5. The number of input and output neurons is equal to the number of active microservices in the cluster, with the input value corresponding to queue depths, and the output value to the probability for a given microservice to initiate a QoS violation. Input neurons are ordered according to the topological application structure, with dependent microservices corresponding to consecutive neurons to capture hotspot patterns in space. Input traces are annotated with the microservices that caused QoS violations in the past.

The choice of DNN architecture is also instrumental to its accuracy. There are three main DNN designs that are popular today: fully connected networks (FC), convolutional neural networks (CNN), and recurrent neural networks (RNN), especially their Long Short-Term Memory (LSTM) class. For Seer to be effective in improving performance predictability, inference needs to occur with enough slack for the cluster manager's action to take effect. Hence, we focus on the more computationally-efficient CNN and LSTM networks. CNNs are especially effective at reducing the dimensionality of large datasets, and finding patterns in space, e.g., in image recognition. LSTMs, on the other hand, are particularly effective at finding patterns in time, e.g., predicting tomorrow's weather based on today's measurements. Signaling QoS violations in a large cluster requires both spatial recognition, namely identifying problematic clusters of microservices whose dependencies cause QoS violations and discarding noisy but non-critical microservices, and temporal recognition, namely using past QoS violations to anticipate future ones. We compare three network designs, a CNN, a LSTM, and a hybrid network that combines the two, using the CNN first to reduce the dimensionality and filter out microservices that do not affect end-to-end performance, and then an LSTM with a *SoftMax* final layer to infer the probability for each microservice to initiate a QoS violation. The architecture of the hybrid network is shown in Fig. 9. Each network is configured using hyperparameter tuning to avoid overfitting, and the final parameters are shown in Table 2.

We train each network on a week's worth of trace data collected on a 20-server cluster running all end-to-end services (for methodology details see Sec. 5) and test it on traces collected on a different week, after the servers had been patched, and the OS had been upgraded.

The quantitative comparison of the three networks is shown in Fig. 10. The CNN is by far the fastest, but also the worst performing, since it is not designed to recognize patterns in time that lead to QoS violations. The LSTM on the other hand is especially effective at capturing load patterns over time, but is less effective at reducing the dimensionality of the original dataset, which makes it prone to false positives due to microservices with many outstanding requests, which are off the critical path. It also incurs higher overheads for inference than the CNN. Finally, Seer correctly anticipates 93.45% of violations, outperforming both networks, for a small increase in inference time compared to LSTM. Given that most resource partitioning decisions take effect after a few 100ms, the inference time for Seer is within the window of opportunity the cluster manager has to take action. More importantly it attributes the QoS violation to the correct microservice, simplifying the cluster manager's task. QoS violations missed by Seer included four random load spikes, and a network switch failure which caused high packet drops.

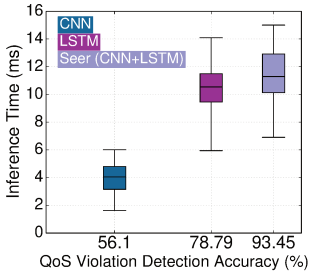

Figure 10. Comparison of DNN architectures.

Out of the five end-to-end services, the one most prone to QoS violations initially was *Social Network*, first, because it has stricter QoS constraints than e.g., *E-commerce*, and second, due to a synchronous and cyclic communication between three neighboring services that caused them to enter a positive feedback loop until saturation. We reimplemented the communication protocol between them post-detection. On the other hand, the service for which QoS violations were hardest to detect was *Media Service*, because of a faulty memory bandwidth partitioning mechanism in one of our servers, which resulted in widely inconsistent memory bandwidth allocations during movie streaming. Since the QoS violation only occurred when the specific streaming microservice was scheduled on the faulty node, it was hard for Seer to collect enough datapoints to signal the violation.

Retraining Seer: By default training happens once, and can be time consuming, taking several hours up to a day for week-long traces collected on our 20-server cluster (Sec. 5 includes a detailed sensitivity study for training time). However, one of the main advantages of microservices is that they simplify frequent application updates, with old microservices often swapped out and replaced by newer modules, or large services progressively broken down to microservices. If the application (or underlying hardware) change significantly, Seer's detection accuracy can be impacted. To adjust to changes in the execution environment, Seer retrains incrementally in the background, using the *transfer learning*-based approach in [80]. Weights from previous training rounds are stored in disk, allowing the model to continue training from where it last left off when new data arrive, reducing the training time by 2-3 orders of magnitude. Even though this approach allows Seer to handle application changes almost in real-time, it is not a long-term solution, since new weights are still polluted by the previous application architecture. When the application changes in a major way, e.g., microservices on the critical path change, Seer also retrains from scratch in the background. While the new network trains, QoS violation detection happens with the incrementally-trained interim model. In Section 5, we evaluate Seer's ability to adjust its estimations to application changes.

4.4 Hardware Monitoring

Once a QoS violation is signaled and a culprit microservice is pinpointed, Seer uses low-level monitoring to identify the reason behind the QoS violation. The exact process depends on whether Seer has access to performance counters.

Private cluster: When Seer has access to hardware events, such as performance counters, it uses them to determine the utilization of different shared resources. Note that even though utilization is a problematic metric for anticipating QoS violations in a large-scale service, once a culprit microservice has been identified, examining the utilization of different resources can provide useful hints to the cluster manager on suitable decisions to avoid degraded performance. Seer specifically examines CPU, memory capacity and bandwidth, network bandwidth, cache contention, and storage I/O bandwidth when prioritizing a resource to adjust. Once the saturated resource is identified, Seer notifies the cluster manager to take action.

Public cluster: When Seer does not have access to performance counters, it instead uses a set of 10 tunable contentious microbenchmarks, each of them targeting a different shared resource [30] to determine resource saturation. For example, if Seer injects the memory bandwidth microbenchmark in the system, and tunes up its intensity without an impact on the co-scheduled microservice's performance, memory bandwidth is most likely not the resource that needs to be adjusted. Seer starts from microbenchmarks corresponding to core resources, and progressively moves to resources further away from the core, until it sees a substantial change in performance when running the microbenchmark. Each microbenchmark takes approximately 10ms to complete, avoiding prolonged degraded performance.

Upon identifying the problematic resource(s), Seer notifies the cluster manager, which takes one of several resource allocation actions, resizing the Docker container, or using mechanisms like Intel's Cache Allocation Technology (CAT) for last level cache (LLC) partitioning, and the Linux traffic control's hierarchical token bucket (HTB) queueing discipline in `qdisc` [17, 62] for network bandwidth partitioning.

4.5 System Insights from Seer

Using learning-based, data-driven approaches in systems is most useful when these techniques are used to gain insight into system problems, instead of treating them as black boxes. Section 5 includes an analysis of the causes behind QoS violations signaled by Seer, including application bugs, poor resource provisioning decisions, and hardware failures. Furthermore, we have deployed Seer in a large installation of the *Social Network* service over the past few months, and its output has been instrumental not only in guaranteeing QoS, but in understanding sources of unpredictable performance, and improving the application design. This has resulted both in progressively fewer QoS violations over time, and a better understanding of the design challenges of microservices.

4.6 Implementation

Seer is implemented in 12KLOC of C,C++, and Python. It runs on Linux and OSX and supports applications in various languages, including all frameworks the end-to-end services are designed in. Furthermore, we provide automated patches for the instrumentation probes for many popular microservices, including `NGINX`, `memcached`, `MongoDB`, `Xapian`, and all Sockshop and *Go-microservices* applications to minimize the development effort from the user's perspective.

Seer is a centralized system; we use master-slave mirroring to improve fault tolerance, with two hot stand-by masters that can take over if the primary system fails. Similarly, the trace database is also replicated in the background.

Security concerns: Trace data is stored and processed unencrypted in Cassandra. Previous work has shown that the sensitivity applications have to different resources can leak information about their nature and characteristics, making them vulnerable to malicious security attacks [27, 36, 50, 52, 84, 88, 89, 94, 99]. Similar attacks are possible using the data and techniques in Seer, and are deferred to future work.

5 Seer Analysis and Validation

5.1 Methodology

Server clusters: First, we use a dedicated local cluster with 20, 2-socket 40-core servers with 128GB of RAM each. Each

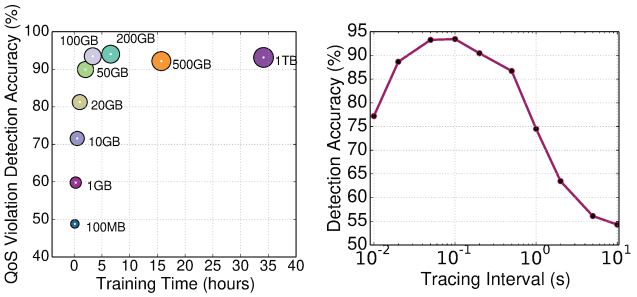

Figure 11. Seer's sensitivity to (a) the size of training datasets, and (b) the tracing interval.

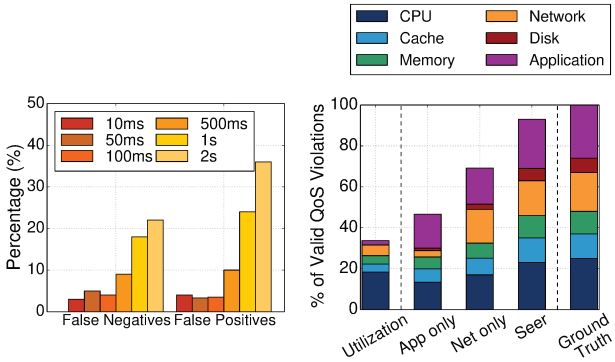

Figure 12. (a) The false negatives and false positives in Seer as we vary the inference window. (b) Breakdown to causes of QoS violations, and comparison with utilization-based detection, and systems with limited instrumentation.

server is connected to a 40Gbps ToR switch over 10Gbe NICs. Second, we deploy the Social Network service to Google Compute Engine (GCE) and Windows Azure clusters with hundreds of servers to study the scalability of Seer.

Applications: We use all five end-to-end services of Table 1. Services for now are driven by open-loop workload generators, and the input load varies from constant, to diurnal, to load with spikes in user demand. In Section 6 we study Seer in a real large-scale deployment of the *Social Network*; in that case the input load is driven by real user traffic.

5.2 Evaluation

Sensitivity to training data: Fig. 11a shows the detection accuracy and training time for Seer as we increase the size of the training dataset. The size of the dots is a function of the dataset size. Training data is collected from the 20-server cluster described above, across different load levels, placement strategies, time intervals, and request types. The smallest training set size (100MB) is collected over ten minutes of the cluster operating at high utilization, while the largest dataset (1TB) is collected over almost two months of continuous deployment. As datasets grow Seer's accuracy increases, leveling off at 100-200GB. Beyond that point accuracy does not further increase, while the time needed for training grows significantly. Unless otherwise specified, we use the 100GB training dataset.

Sensitivity to tracing frequency: By default the distributed tracing system instantaneously collects the latency of every single user request. Collecting queue depth statistics, on the other hand, is a per-microservice iterative process. Fig. 11b shows how Seer's accuracy changes as we vary the frequency with which we collect queue depth statistics. Waiting for a long time before sampling queues, e.g., > 1*s*, can result in undetected QoS violations before Seer gets a chance to process the incoming traces. In contrast, sampling queues very frequently results in unnecessarily many inferences, and runs the risk of increasing the tracing overhead. For the remainder of this work, we use 100*ms* as the interval for measuring queue depths across microservices.

False negatives & false positives: Fig. 12a shows the percentage of false negatives and false positives as we vary the prediction window. When Seer tries to anticipate QoS violations that will occur in the next 10-100ms both false positives and false negatives are low, since 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. However, given that inference takes several milliseconds and more importantly, applying corrective actions to avoid QoS violations takes 10-100s 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.

Comparison of debugging systems: Fig. 12b compares Seer with a utilization-based performance debugging system that uses resource saturation as the trigger to signal a QoS violation, and two systems that only use a fraction of Seer's instrumentation. `App-only` exclusively uses queues measured via application instrumentation (not network queues), while `Network-only` uses queues in the NIC, and ignores application-level queueing. We also show the ground truth for the total number of upcoming QoS violations (96 over a two-week period), and break it down by the reason that led to unpredictable performance.

A large fraction of QoS violations are due to application-level inefficiencies, including correctness bugs, unnecessary synchronization and/or blocking behavior between microservices (including two cases of deadlocks), and misconfigured iptables rules, which caused packet drops. An equally large fraction of QoS violations are due to compute contention, followed by contention in the network, cache and memory

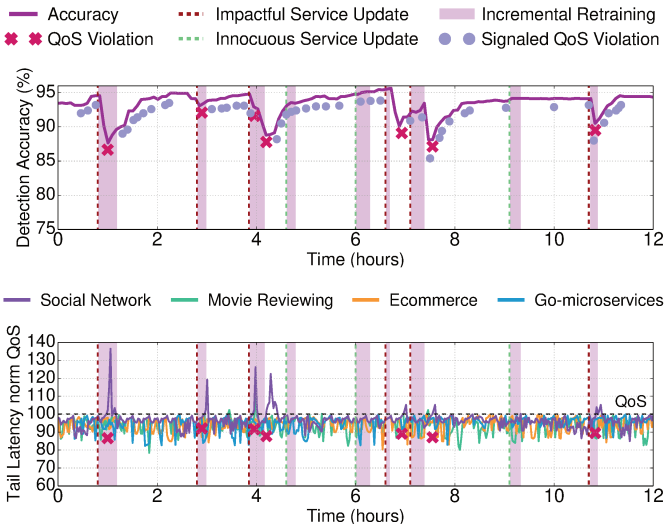

Figure 13. Seer retraining incrementally after each time the *Social Network* service is updated.

contention, and finally disk. Since the only persistent microservices are the back-end databases, it is reasonable that disk accounts for a small fraction of overall QoS violations.

Seer accurately follows this breakdown for the most part, only missing a few QoS violations due to random load spikes, including one caused by a switch failure. The `App-only` system correctly identifies application-level sources of unpredictable performance, but misses the majority of system-related issues, especially in uncore resources. On the other hand, `Network-only` correctly identifies the vast majority of network-related issues, as well as most of the core- and uncore-driven QoS violations, but misses several application-level issues. The difference between `Network-only` and Seer is small, suggesting that one could omit the application-level instrumentation in favor of a simpler design. While this system is still effective in capturing QoS violations, it is less useful in providing feedback to application developers on how to improve their design to avoid QoS violations in the future. Finally, the utilization-based system behaves the worst, missing most violations not caused by CPU saturation.

Out of the 89 QoS violations Seer detects, it notifies the cluster manager early enough to avoid 84 of them. The QoS violations that were not avoided correspond to application-level bugs, which cannot be easily corrected online. Since this is a private cluster, Seer uses utilization metrics and performance counters to identify problematic resources.

Retraining: Fig. 13 shows the detection accuracy for Seer, and the tail latency for each end-to-end service, over a period of time during which *Social Network* is getting frequently and substantially updated. This includes new microservices being added to the service, such as the ability to place an order from an ad using the `orders` microservice of *E-commerce*, or the back-end of the service changing from `MongoDB` to `Cassandra`, and the front-end switching from `nginx` to the node.js front-end of *E-commerce*. These are changes that fundamentally affect the application's behavior, throughput, latency, and bottlenecks. The other services remain unchanged during this period (*Banking* was not active during this time, and is omitted from the graph). Blue dots denote correctly-signaled upcoming QoS violations, and red × denote QoS violations that were not detected by Seer. All unidentified QoS violations coincide with the application being updated. Shortly after the update Seer incrementally retrains in the background, and starts recovering its accuracy until another major update occurs. Some of the updates have no impact on either performance or Seer's accuracy, either because they involve microservices off the critical path, or because they are insensitive to resource contention.

The bottom figure shows that unidentified QoS violations indeed result in performance degradation for *Social Network*, and in some cases for the other end-to-end services, if they are sharing physical resources with *Social Network*, on an oversubscribed server. Once retraining completes the performance of the service(s) recovers. The longer Seer trains on an evolving application, the more likely it is to correctly anticipate its future QoS violations.

6 Large-Scale Cloud Study

6.1 Seer Scalability

We now deploy our *Social Network* service on a 100-instance dedicated cluster on Google Compute Engine (GCE), and use it to service real user traffic. The application has 582 registered users, with 165 daily active users, and has been deployed for a two-month period. The cluster on average hosts 386 single-concerned containers (one microservice per container), subject to some resource scaling actions by the cluster manager, based on Seer's feedback.

Accuracy remains high for Seer, consistent with the small-scale experiments. Inference time, however, increases substantially from 11.4ms for the 20-server cluster to 54ms for the 100-server GCE setting. Even though this is still sufficient for many resource allocation decisions, as the application scales further, Seer's ability to anticipate a QoS violation within the cluster manager's window of opportunity diminishes.

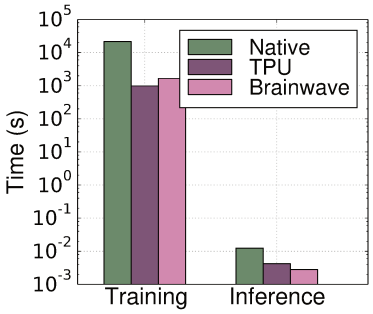

Figure 14. Seer training and inference with hardware acceleration.

Over the past year multiple public cloud providers have exposed hardware acceleration offerings for DNN training and

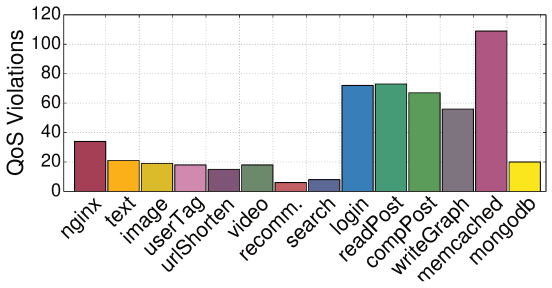

Figure 15. QoS violations each microservice in *Social Network* is responsible for.

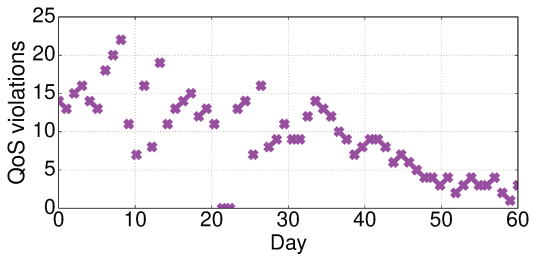

Figure 16. QoS violations each microservice in *Social Network* is responsible for.

inference, either using a special-purpose design like the Tensor Processing Unit (TPU) from Google [55], or using reconfigurable FPGAs, like Project Brainwave from Microsoft [25]. We offload Seer's DNN logic to both systems, and quantify the impact on training and inference time, and detection accuracy [1]. Fig. 14 shows this comparison for a 200GB training dataset. Both the TPU and Project Brainwave dramatically outperform our local implementation, by up to two orders of magnitude. Between the two accelerators, the TPU is more effective in training, consistent with its design objective [55], while Project Brainwave achieves faster inference. For the remainder of the paper, we run Seer on TPUs, and host the *Social Network* service on GCE.

6.2 Source of QoS Violations

We now examine which microservice is the most common culprit for a QoS violation. Fig. 15 shows the number of QoS violations caused by each service over the two-month period. The most frequent culprits by far are the in-memory caching tiers in `memcached`, and Thrift services with high request fanout, such as `composePost`, `readPost`, and `login`. `memcached` is a justified source of QoS violations, since it is on the critical path for almost all query types, and it is additionally very sensitive to resource contention in compute and to a lesser degree cache and memory. Microservices with high fanout are also expected to initiate QoS violations, as they have to synchronize multiple inbound requests before proceeding. If processing for any incoming requests is delayed, end-to-end performance is likely to suffer. Among these QoS violations, most of `memcached`'s violations were caused by resource contention, while violations in Thrift services were caused by long synchronization times.

6.3 Seer's Long-Term Impact on Application Design

Seer has now been deployed in the *Social Network* cluster for over two months, and in this time it has detected 536 upcoming QoS violations (90.6% accuracy) and avoided 495 (84%) of them. Furthermore, by detecting recurring patterns that lead to QoS violations, Seer has helped the application developers better understand bugs and design decisions that lead to hotspots, such as microservices with a lot of back-and-forth communication between them, or microservices forming cyclic dependencies, or using blocking primitives. This has led to a decreasing number of QoS violations over the two month period (seen in Fig. 16), as the application progressively improves. In days 22 and 23 there was a cluster outage, which is why the reported violations are zero. Systems like Seer can be used not only to improve performance predictability in complex cloud systems, but to help users better understand the design challenges of microservices, as more services transition to this application model.

[1]Before running on TPUs, we reimplemented our DNN in Tensorflow. We similarly adjust the DNN to the currently-supported designs in Brainwave.

7 Conclusions

Cloud services increasingly move away from complex monolithic designs, and adopt the model of specialized, loosely-coupled microservices. We presented Seer, a data-driven cloud performance debugging system that leverages practical learning techniques, and the massive amount of tracing data cloud systems collect to proactively detect and avoid QoS violations. We have validated Seer's accuracy in controlled environments, and evaluated its scalability on large-scale clusters on public clouds. We have also deployed the system in a cluster hosting a social network with hundreds of users. In all scenarios, Seer accurately detects upcoming QoS violations, improving responsiveness and performance predictability. As more services transition to the microservices model, systems like Seer provide practical solutions that can navigate the increasing complexity of the cloud.

Acknowledgements

We sincerely thank Christos Kozyrakis, Balaji Prabhakar, Mendel Rosenblum, Daniel Sanchez, Ravi Soundararajan, the academic and industrial users of the benchmark suite, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported by NSF CAREER Award CCF-1846046, NSF grant CNS-1422088, a Facebook Faculty Research Award, a VMWare Research Award, a John and Norma Balen Sesquicentennial Faculty Fellowship, and donations from GCE, Azure, and AWS.

References

[1] [n. d.]. Apache Thrift. https://thrift.apache.org.

[2] [n. d.]. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[3] [n. d.]. Golang Microservices Example. https://github.com/harlow/go-micro-services.

[4] [n. d.]. Messaging that just works. https://www.rabbitmq.com/.

[5] [n. d.]. MongoDB. https://www.mongodb.com.

[6] [n. d.]. NGINX. https://nginx.org/en.

[7] [n. d.]. SockShop: A Microservices Demo Application. https://www.weave.works/blog/sock-shop-microservices-demo-application.

[8] [n. d.]. Zipkin. http://zipkin.io.

[9] 2016. The Evolution of Microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference.

[10] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. [n. d.]. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *Proceedings of OSDI, 2016.*

[11] Adrian Cockroft [n. d.]. Microservices Workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[12] Hyunwook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. [n. d.]. CloudSight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2017.

[13] Luiz Barroso. [n. d.]. Warehouse-Scale Computing: Entering the Teenage Decade. *ISCA Keynote, SJ, June 2011* ([n. d.]).

[14] Luiz Barroso and Urs Hoelzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers.

[15] Robert Bell, Yehuda Koren, and Chris Volinsky. 2007. *The BellKor 2008 Solution to the Netflix Prize*. Technical Report.

[16] Leon Bottou. [n. d.]. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*. Paris, France, 2010.

[17] Martin A. Brown. [n. d.]. Traffic Control HOWTO. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[18] Cassandra [n. d.]. Apache Cassandra. http://cassandra.apache.org/.

[19] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-scale Acceleration Architecture. In *MICRO*. IEEE Press, Piscataway, NJ, USA, Article 7, 13 pages.

[20] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*.

[21] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-efficient Hardware Accelerators for Machine Learning. *Commun. ACM* 59, 11 (Oct. 2016), 105–112. https://doi.org/10.1145/2996864

[22] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622.

[23] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. 2007. Comparison of the Three CPU Schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.* 35, 2 (Sept. 2007), 42–51.

[24] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. 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 (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 217–231.

[25] Eric S. Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian M. Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamás Juhász, Kara Kagi, Ratna Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven K. Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[26] Guilherme Da Cunha Rodrigues, Rodrigo N. Calheiros, Vinicius Tavares Guimaraes, Glederson Lessa dos Santos, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville, Liane Margarida Rockenbach Tarouco, and Rajkumar Buyya. 2016. Monitoring of Cloud Computing Environments: Concepts, Solutions, Trends, and Future Directions. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. ACM, New York, NY, USA, 378–383.

[27] Marwan Darwish, Abdelkader Ouda, and Luiz Fernando Capretz. [n. d.]. Cloud-based DDoS attacks and defenses. In *Proc. of i-Society*. Toronto, ON, 2013.

[28] Jeffrey Dean and Luiz Andre Barroso. [n. d.]. The Tail at Scale. In *CACM, Vol. 56 No. 2*.

[29] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. [n. d.]. QoS-Aware Admission Control in Heterogeneous Datacenters. In *Proceedings of the International Conference of Autonomic Computing (ICAC)*. San Jose, CA, USA, 2013.

[30] Christina Delimitrou and Christos Kozyrakis. [n. d.]. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013.

[31] Christina Delimitrou and Christos Kozyrakis. [n. d.]. 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.

[32] Christina Delimitrou and Christos Kozyrakis. [n. d.]. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *ACM Transactions on Computer Systems (TOCS), Vol. 31 Issue 4*. December 2013.

[33] Christina Delimitrou and Christos Kozyrakis. [n. d.]. 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.

[34] Christina Delimitrou and Christos Kozyrakis. [n. d.]. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of ASPLOS*. Salt Lake City, 2014.

[35] Christina Delimitrou and Christos Kozyrakis. 2016. 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)*.

[36] Christina Delimitrou and Christos Kozyrakis. 2017. 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)*.

[37] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's Law for Tail Latency. In *Communications of the ACM (CACM)*.

[38] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*.

[39] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/2749469.2750389

[40] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66.

[41] Brad Fitzpatrick. [n. d.]. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*.

[42] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 20–20.

[43] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2*.

[44] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He, and Christina Delimitrou. 2018. Seer: Leveraging Big Data to Navigate the Complexity of Cloud Debugging. In *Proceedings of the Tenth USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[45] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[46] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. [n. d.]. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proc. of NSDI*. 2018.

[47] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. [n. d.]. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *Proceedings of IISWC*. Boston, MA, 2007, 10. https://doi.org/10.1109/IISWC.2007.4362193

[48] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. [n. d.]. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proceedings of CNSM*. Niagara Falls, ON, 2010.

[49] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. [n. d.]. Fundamentals of Queueing Theory. In *Wiley Series in Probability and Statistics, Book 627*. 2011.

[50] Sanchika Gupta and Padam Kumar. [n. d.]. VM Profile Based Optimized Network Attack Pattern Detection Scheme for DDOS Attacks in Cloud. In *Proc. of SSCC*. Mysore, India, 2013.

[51] Ben Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. [n. d.]. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI*. Boston, MA, 2011.

[52] Jingwei Huang, David M. Nicol, and Roy H. Campbell. [n. d.]. Denial-of-Service Threat to Hadoop/YARN Clusters with Multi-Tenancy. In *Proc. of the IEEE International Congress on Big Data*. Washington, DC, 2014.

[53] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. [n. d.]. On the Performance Variability of Production Cloud Services. In *Proceedings of CCGRID*. Newport Beach, CA, 2011.

[54] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of WWW*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 469–478.

[55] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12.

[56] Harshad Kasture and Daniel Sanchez. 2014. 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)*.

[57] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *Proc. of IISWC*.

[58] Yaakoub El Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. [n. d.]. Exploring the performance fluctuations of hpc workloads on clouds. In *Proceedings of CloudCom*. Indianapolis, IN, 2010.

[59] Krzysztof C. Kiwiel. [n. d.]. Convergence and efficiency of subgradient methods for quasiconvex minimization. In *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001*.

[60] Jacob Leverich and Christos Kozyrakis. [n. d.]. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. of EuroSys*. 2014.

[61] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. [n. d.]. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Minneapolis, MN, 2014.

[62] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. [n. d.]. Heracles: Improving Resource Efficiency at Scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[63] Dave Mangot. [n. d.]. EC2 variability: The numbers revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_re%vealed.

[64] Jason Mars and Lingjia Tang. [n. d.]. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of ISCA*. Tel-Aviv, Israel, 2013.

[65] Jason Mars, Lingjia Tang, and Robert Hundt. 2011. Heterogeneity in "Homogeneous"; Warehouse-Scale Computers: A Performance Opportunity. *IEEE Comput. Archit. Lett.* 10, 2 (July 2011), 4.

[66] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. [n. d.]. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of MICRO*. Porto Alegre, Brazil, 2011.

[67] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*. 319–330.

[68] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. [n. d.]. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of ICAC*. Jacksonville, FL, 2007.

[69] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. [n. d.]. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proceedings of EuroSys*. Paris,France, 2010.

[70] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. [n. d.]. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of ICAC*. San Jose, CA, 2013.

[71] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. [n. d.]. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of ATC*. San Jose, CA, 2013.

[72] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. [n. d.]. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *Lecture Notes on Cloud Computing*. Volume 34, p.115-131, 2010.

[73] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. [n. d.]. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of SOSP*. Farminton, PA, 2013.

[74] Xue Ouyang, Peter Garraghan, Renyu Yang, Paul Townend, and Jie Xu. [n. d.]. Reducing Late-Timing Failure at Scale: Straggler Root-Cause Analysis in Cloud Datacenters. In *Proceedings of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2016.

[75] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proc. of the 41st Intl. Symp. on Computer Architecture*.

[76] Suhail Rehman and Majd Sakr. [n. d.]. Initial Findings for provisioning variation in cloud computing. In *Proceedings of CloudCom*. Indianapolis, IN, 2010.

[77] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. [n. d.]. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of SOCC*. 2012.

[78] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* (2010), 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[79] rightscale [n. d.]. Rightscale. https://aws.amazon.com/solution-providers/isv/rightscale.

[80] S. Sarwar, A. Ankit, and K. Roy. [n. d.]. Incremental Learning in Deep Convolutional Neural Networks Using Partial Network Sharing. In *arXiv preprint arXiv:1712.02719*.

[81] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings VLDB Endow.* 3, 1-2 (Sept. 2010), 460–471.

[82] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. [n. d.]. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*. Prague, 2013.

[83] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. [n. d.]. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of SOCC*. Cascais, Portugal, 2011.

[84] David Shue, Michael J. Freedman, and Anees Shaikh. [n. d.]. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proc. of OSDI*. Hollywood, CA, 2012.

[85] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[86] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. [n. d.]. Distributed Resource Management Across Process Boundaries. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*. Santa Clara, CA, 2017.

[87] torque [n. d.]. Torque Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/.

[88] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. [n. d.]. Scheduler-based Defenses against Cross-VM Side-channels. In *Proc. of the 23rd Usenix Security Symposium*. San Diego, CA, 2014.

[89] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. [n. d.]. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proc. of the 24th USENIX Security Symposium (USENIX Security)*. Washington, DC, 2015.

[90] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. [n. d.]. DejaVu: accelerating resource allocation in virtualized environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK, 2012.

[91] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. [n. d.]. Root cause analysis of anomalies of multitier services in public clouds. In *Proceedings of the IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. 2017.

[92] Ian H. Witten, Eibe Frank, and Geoffrey Holmes. [n. d.]. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[93] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 619–634.

[94] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. [n. d.]. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proc. of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW)*. Chicago, IL, 2011.

[95] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. [n. d.]. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of ISCA*. 2013.

[96] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. [n. d.]. Profiling Network Performance for Multi-tier Data Center Applications. In *Proceedings of NSDI*. Boston, MA, 2011, 14.

[97] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs. In *Proceedings of APSLOS*. ACM, New York, NY, USA, 489–502.

[98] Yinqian Zhang and Michael K. Reiter. [n. d.]. Duppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. of CCS*. Berlin, Germany, 2013.

[99] Mark Zhao and G. Edward Suh. [n. d.]. FPGA-Based Remote Power Side-Channel Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*. May 2018.