A Cloud-Scale Characterization of Remote Procedure Calls

Korakit Seemakhupt
University of Virginia

Brent E. Stephens
Google and University of Utah

Samira Khan
Google and University of Virginia

Sihang Liu
University of Waterloo

Hassan Wassel
Google

Soheil Hassas Yeganeh
Google

Alex C. Snoeren
Google and UC San Diego

Arvind Krishnamurthy
Google and University of Washington

David E. Culler
Google

Henry M. Levy
Google and University of Washington

Abstract

The global scale and challenging requirements of modern cloud applications have led to the development of complex, widely distributed, service-oriented applications. One enabler of such applications is the remote procedure call (RPC), which provides location-independent communication and hides the myriad of cloud communication complexities and requirements within the RPC stack. Understanding RPCs is thus one key to understanding the behavior of cloud applications. While there have been numerous studies of RPCs in distributed systems, as well as attempts to optimize RPC overheads with both software and hardware, there is still a lack of knowledge about the characteristics of RPCs "in the wild" in the modern cloud environment.

To address this gap, we present, to the best of our knowledge, the first large-scale fleet-wide study of RPCs. Our study is conducted at Google, where we measured the infrastructure supporting Google's user-facing, billion-user web services, such as Google Search, Gmail, Maps, and YouTube, and the information and data management systems that support them. To carry out the study, we examined over 10,000 different RPC methods sampled from over one billion traces, along with statistics collected every 30 minutes over a period of nearly two years. Among other things, we consider the volume, throughput and growth rate of RPCs in the datacenter, the latency of RPCs and their components (the "RPC latency tax"), and the structure of RPC call chains. Our analysis shows that the characteristics, scope and complexity of RPCs at hyperscale differ significantly from the assumptions made in prior research. Overall, our work provides new insights into RPC usage and characteristics at the largest scale and motivates further research on optimizing the diverse behavior of this crucial communication mechanism.

CCS Concepts: • **Networks** → **Network measurement**; • **Computer systems organization** → **Cloud computing**.

Keywords: Remote procedure call, Cloud computing, Distributed computing, Communications systems

ACM Reference Format:
Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613156

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613156

1 Introduction

Modern cloud computing plays an increasingly critical role in today's Web services, data storage, analytics, and emerging applications like AI/ML and AR/VR. A key advantage of the cloud is the ability for applications to dynamically *scale out* to meet changing workloads. Cloud applications can achieve high availability, fault isolation, and easier maintenance, in part, through deployment and replication of computation and data across datacenters and geographical regions.

To scale out, a single application is divided into multiple distributed communicating services. Currently, the standard inter-communication layer for cloud services is Remote Procedure Call (RPC) [10, 12, 13, 48, 73, 76]. RPC greatly simplifies application development in a distributed system. In particular, a function call to a remote machine looks similar to a function call within the same application [14], and complexities such as connection management, network protocols, parameter marshalling/demarshalling, encryption/decryption, and thread scheduling are handled by the RPC stack, which is typically implemented as a userspace library [44, 66, 68].

In effect, RPC is the fundamental control structure that defines the flow of both computation and data in a distributed system.[1] For this reason, RPC is a valuable lens for viewing the behavior and performance of modern cloud-based widely-distributed applications. While prior work has analyzed the behavior of datacenter networks [7, 11, 60] and the microarchitectural CPU behavior of datacenter applications [42], there is little analysis and understanding of RPCs *at cloud scale*. This is a significant limitation, because RPCs generate the work performed by the network, the CPUs, and the storage systems. For example, it is necessary to consider the RPC communication graphs to understand the dependencies across network flows, e.g., co-flows [21].

This paper presents, to our knowledge, the first analysis of RPCs from geo-distributed applications running across a fleet of datacenters. Our study is unique in its massive scale: we evaluate RPC calls in Google's internal-application environment collected over a period of 23 months, using multiple tools and data sources describing RPC characteristics and behavior. For example, we examine counters collected every 30 minutes from over 10,000 different unique RPC methods (procedures) running in 100s of different clusters, and we analyze over 722 billion RPC samples collected from a single day. This data, which we describe more in following sections, allows us to discover key properties such as the volume of RPCs and its growth over time, the contributions of different RPC components to latency, and the depth and structure of RPC call trees. Together, these different analysis methodologies allow us to conduct a thorough and in-depth analysis of the behavior of RPCs and global-scale applications in one of the world's largest hyperscalers.

In more detail, the contributions of this paper include the following:

- We show that the use of RPC is increasing rapidly, indicating the importance of RPC optimization.
- We characterize the latency, frequency, size, and nested hierarchy of over 10,000 distinct RPC methods, showing that their characteristics differ significantly from those assumed by prior work. For example, on average, RPCs in our environment operate at millisecond timescales and kilobyte sizes with relatively deep nested hierarchies.
- We analyze the latency components in RPC completion time and find that the RPC latency bottleneck differs from prior assumptions. On average, the majority of this time is spent on application processing, but tail latency is dominated by the RPC latency tax, i.e., the time spent on queues, RPC processing, and network transfers.
- We characterize the latency components in eight top services and show the latency variation within and across physically co-located clusters of servers distributed across datacenters. We find that high server and memory utilization leads to high variation within clusters, while network latency dominates the variation across clusters.
- Finally, we analyze the CPU cycle variation in processing RPCs across the fleet and find that there is a significant variance in CPU cycles used for RPCs, indicating the opportunity for better load-balancing.

Overall, our characterization of RPCs provides a perspective on the scope, complexity, and variance of the cloud services that implement some of the world's largest-scale web applications. We also hope to motivate future research on designing effective optimizations of applications and RPCs.

The remainder of the paper is organized as follows. Section 2 explores the scope and complexity of RPCs in Google's hyperscale environment. In Section 3, we examine the RPC latency and its major components that affect RPC completion time and variance. Section 4 considers resource utilization in RPC services across the datacenter, focusing on the opportunity for load-balancing. We discuss implications of our research in Section 5, present previous work in Section 6, and conclude in Section 7.

2 Characteristics of RPCs at Hyperscale

This section analyzes the properties of the RPCs used to implement Google's external services and internal data processing systems in aggregate across its datacenters. For each RPC that we analyze, we measure the behavior of the RPC stack and the RPC handler (i.e., the invoked method). Specifically, we measure RPCs from Google's first-party commercial products, including both user-facing web services (e.g., Google Search, Google Maps, Gmail, and YouTube), as well as the massive internal services and data processing systems that support these web services (e.g., Spanner [22], BigQuery [26], Bigtable [17], F1 [64], GFS [30], and Chubby [15]). Our study does *not* include Google's external Cloud product (GCP) or RPCs issued by its customers.

These Google internal services are all built with similar design patterns. In particular, they are widely distributed, e.g., they consist of many replicated parallel tasks running in multiple geo-distributed datacenters to ensure high availability. Nearly all of the services we study are built with the Stubby RPC stack [12], a Google-internal RPC library that provides features similar to gRPC. To operate at a massive scale, many of these services utilize a partition/aggregate design pattern [24, 64]. Data for these services is stored remotely in a network filesystem that replicates blocks across multiple machines in different datacenters for fault tolerance [30]. In these applications, computation flows from front-end applications to back-end services and then to the network filesystem. The individual RPCs that compose these applications can handle large amounts of data and are often computationally intensive. As such, these applications are based on a service-oriented architecture rather than a microservice architecture

[1] RDMA [1, 25, 34, 70, 77] systems are increasingly used along with RPC for data movement, but we focus here on RPC.

in that they are not decomposed into the smallest possible functional units. While we expect to see an increase in microservices in the future, the rapid growth in data collection and processing may well offset that trend.

Because this is the first study of its kind, it is difficult to know the extent to which the behavior of Google's internal applications generalizes to the behavior of the distributed systems used in other hyperscalars. Different cloud and communications architectures can lead to different design decisions, and it would be interesting to understanding other designs and their implications. However, this study is an important first step toward understanding RPC in the context of massive scale applications and their supporting services.

2.1 Methodology

In our analysis, we primarily focus on the most common RPC stacks at Google: Stubby [12] and gRPC [44]. Our analysis is performed using three different Google-internal monitoring tools, Monarch [4], Dapper [65], and GWP [58]. We particularly focus on the distribution of RPC completion time, size, and depth, as prior works propose optimizations based on these RPC properties [8, 40, 41, 46, 47, 53, 56, 72, 74].

Monarch is a monitoring database that performs periodic sampling of various metrics exported by individual application instances (tasks). These samples provide distributions of RPC behavior across various levels and dimensions of aggregation, e.g., per-cluster or network-traffic class.[2] Not all metrics in this database have the same retention policies. Some metrics are retained for 700 days with metric samples every 30 minutes. Over shorter time scales like the past 30 days, it is possible to get samples with shorter windows, e.g., one every minute. Our results are based on metrics from a 700-day period between December 2020 and November 2022. We use this timescale of 700 days primarily to observe changes in RPCs over time.

Aggregate time-series data does not provide complete information about the behavior of individual RPCs, so we employ Dapper to conduct some aspects of our analysis. Dapper is an internal service that collects samples of entire RPC trees (traces), where each node in the tree represents an individual child RPC call [65]. For each RPC call, the service collects information about the latency of various components, including the time spent in the client, server, and network. For RPCs that make nested RPC calls, the time of the nested calls is included in the application processing time of the parent RPC. From the RPC handler's perspective, waiting for application processing and waiting for the responses from nested RPCs are the same, as the nesting is invisible to the caller. Similarly, we exclude the latency of error RPCs from the measurement. However, the caller of erroneous RPCs needs to handle errors (unless canceled by the caller), and the latency of these callers is included in the latency measurement. Note that TCP retransmissions are not counted as errors.

[2]The sampling omits some RPC classes, such as streaming RPCs that are used for some bulk-data transfers.

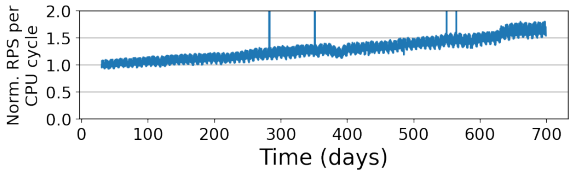


Figure 1. Normalized RPS per CPU cycles consumed over time.

Our analysis that utilizes the Dapper tracing service focuses on RPCs from a single day. When we use tracing to look at different RPC methods, we only consider methods with at least 100 samples so that the 99th percentile is well defined.

We use GWP (Google-Wide Profiling) [58] to study the number of CPU cycles spent processing RPCs in our fleet. GWP collects daily CPU profiles of sampled application execution, and these profiles can be used to identify RPC cycles.

2.2 Why is RPC Evaluation Important?

We evaluate the growth rate of RPCs to demonstrate the importance of RPC and its efficiency in the cloud. Equally important, RPC gives us insight into the organization of a widely distributed system and its dynamic functioning. Figure 1 shows the number of RPCs per second (RPS) in our fleet *divided by the number of CPU cycles consumed* over the 700-day period. The daily RPS/CPU value in this figure is normalized to the first day of our observation, which shows the growth rate of the ratio.

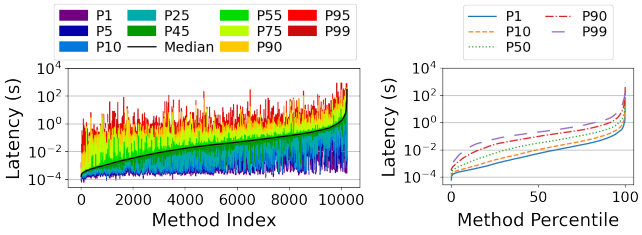
From Figure 1, RPC throughput relative to CPU cycle utilization is increasing at an annual rate of approximately 30%, for a total increase of 64% over the measurement interval. That is, RPC usage is growing *faster* than compute in our cloud. This change is the result of two trends. The first is that increasing the hardware optimization of the RPC stack over time has reduced the CPU cycle cost of each RPC invocation. The second is that the growth of microservice-based design [2, 3, 59] is reducing the number of CPU cycles consumed per RPC, and this is likely to accelerate this trend even more in the future.

This growth rate puts a tremendous demand on network and compute resources, creating major challenges to sustaining this growth.

In the rest of this section we analyze high-level RPC characteristics to understand the basic properties of RPCs and RPC-based services. The deeper analysis in future sections will help to expose potential optimization and acceleration opportunities for cloud-based RPC systems.

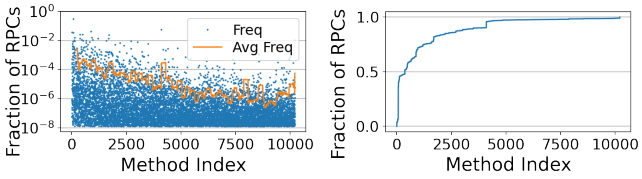
2.3 Not all RPCs are the same.

We analyze the distribution of completion times and the popularity of common RPC methods in order to understand the variance in the timescale on which they operate. Figure 2a shows, for our 10K methods, the RPC completion time (RCT)



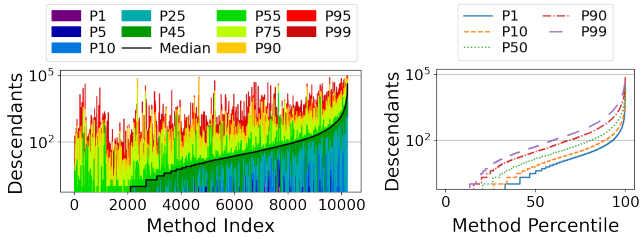
(a) RPC Latency Distribution Heatmap **(b) RPC Latency CDF**

Figure 2. Per-Method RPC latency, sorted by median latency.



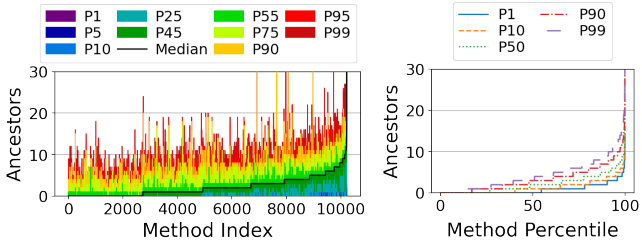
(a) RPC method frequency **(b) CDF of method frequencies**

Figure 3. Per-Method RPC frequency, sorted by median latency.



(a) Descendants sorted in increasing order **(b) CDF**

Figure 4. Per-Method Number of Descendants



(a) Per-method number of ancestors sorted by median in increasing order **(b) CDF of ancestor count**

Figure 5. Per-Method RPC Ancestors.

per method, sorted by the median latency. The RPC completion time includes all latency from the client's RPC invocation until the client receives the response; this includes execution of the method on the server. We show the latency distributions of each method as a heatmap, where brighter colors (e.g., red) indicate P90+ tail latencies, and cooler colors (e.g., blue) indicate P10- latencies of each method. Figure 2b shows a CDF of the tail latencies of different methods.

These figures show that there is significant variance in the amount of time taken to process RPCs across different methods. Most methods are capable of completing RPCs within hundreds of microseconds. For 90% of the methods, the 1st percentile latency is 657 μs or less. However, services often operate at the millisecond scale in our fleet. 90% of the services have a median latency that is 10.7 ms or greater.

At the tail, almost all methods have slow RPCs that operate on the scale of milliseconds. Over 99.5% of methods have a P99 latency of 1 ms or greater; 50% of methods have a P99 latency of 225 ms or greater. Further, the slowest methods are even slower. Of the slowest 5% of methods, the P1 and P99 latencies are 166 ms and 5 s or greater, respectively. We conclude that there is a major variation in the latency of hyperscale RPCs, and it ranges from hundreds of microseconds to seconds.

The latencies of hyperscale RPCs are significantly higher than what has been assumed in recent RPC benchmarks [29] and RPC optimization [35, 38, 41, 46, 72] works. Most of these studies assumed micro-second level RPCs and optimized the RPC stack for better performance. It is clear that we need to better understand the latency bottlenecks in hyperscalar RPCs; we dive deeper into latency components and variation of RPCs in Section 3.

Given the large variation in method latency, we also investigate the popularity of RPC methods. Knowing the popularity distribution tells us the expected benefit of optimizing a small number of methods. Figure 3a shows the popularity (relative frequency) of all 10K methods, and Figure 3b shows the popularity CDF; both graphs are again sorted in latency order.

From Figure 3a we see that not all RPC methods are equally popular, but in particular, many of the low-latency methods (on the left) are extremely popular. In fact, the 100 lowest latency RPC methods account for 40% of all RPC calls. As an extreme point, the "Write" RPC for the Network Disk alone accounts for 28% of all RPC calls.

Sorting by popularity rather than latency (not shown) gives another view of the skew. The 10 most popular methods account for 58% of all calls, and the top-100 account for 91% of all calls. While many long-latency methods may be less frequent, collectively they consume more resources than shorter, more popular methods. The slowest 1000 RPC methods account for only 1.1% of all calls, but they take 89% of the total RPC time. Clearly not all RPCs are the same, and we should target different services and methods for optimization depending on our goals.

2.4 Nested RPCs are Wider than Deep

The distributed nature of service-oriented applications in the fleet results in nested RPC call graphs, where each RPC can fan out to multiple child RPCs. To better understand the shape

of nested RPC calls, we analyze the number of descendants and ancestors of different RPC methods. Looking at the number of descendants shows the scale of distributed computation performed by an RPC, and the number of ancestors provides insights into how the properties of RPCs change as they get deeper into the call graph of a root RPC.

Figure 4 plots the number of descendants for different RPC methods. Half of RPC methods have a median of 13 or fewer descendants. On the other hand, the descendant tail can be quite large: 90% of RPC methods have P90 and P99 descendant counts of over 105 and 1155, respectively.

Figure 5 shows the number of ancestors for each invoked method, *i.e.*, the return distance from a called method to the root RPC in the tree. Compared to the number of descendants, the number of ancestors for a given invocation is much smaller, implying that the typical RPC call tree is wider than it is deep. For example, half of the methods have fewer than 10 ancestors at 99$^{\text{th}}$ percentile.

In early RPC systems, calls typically went to simple, discrete services. However, from our measurements, RPCs in the cloud environment may invoke general computations that include complex call trees and nested RPCs. Understanding this call structure has important implications, e.g., in creating benchmarks for hyperscale services that can accurately represent the shape of complex, nested RPC call graphs.

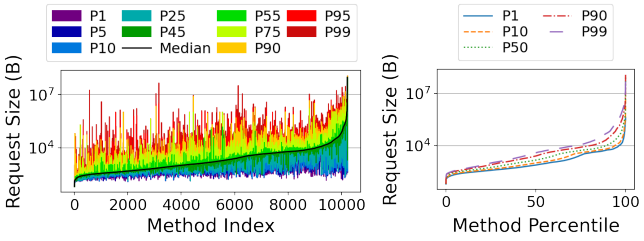
As a point of comparison, Luo *et al.* [48] performed an analysis of RPC call graphs from more than 20,000 microservices at Alibaba (Figure 3). In both this study and the Alibaba study, the call graphs are wider than they are deep. There is a heavy tail many times larger than the median, and the call tree depths are also similar at both the median and the tail. The biggest difference is that the RPC methods that we study have a larger number of descendants, especially at the tail.

Huye *et al.* [37] report properties about the number of service blocks in the request workflows for a few of Meta's internal applications, and they similarly find that these service graphs are much wider than they are deep. Their P99 depth ranges from 5–6 and their maximum depth ranges from 9–19, and this is similar to our findings. Their median total number of blocks per trace ranges from 2–498, and the P99 ranges from ∼1K–10K, and there are RPC methods that we study that also have similarly sized RPC trees.

Gan *et al.* [29] report the depth and size of the service graphs used by the applications in the DeathStarBench (DSB) benchmark. The applications service graph depths range from 3–9, and the total number of services in an application ranges from 21–41. These service graphs are similar in depth to those of the methods that we studied, but the total size of these graphs are smaller, especially at the tail.

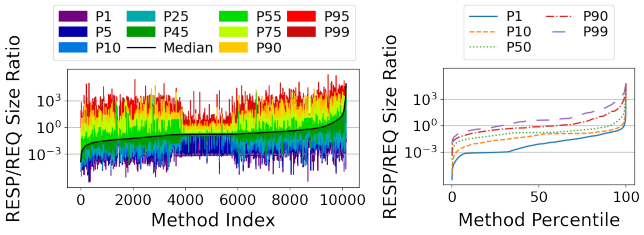
2.5 RPC Size Matters

Characterizing flow sizes and their heavy tail in datacenter workloads has been important in shaping the design of traffic engineering in datacenter networks. At the RPC layer, the individual RPC is the smallest unit that can be load balanced. As such, it is also important to understand the distribution of RPC sizes. For example, knowledge of the RPC size distribution is needed to evaluate changes to how RPCs are mapped onto network flows.



(a) Request Sizes. **(b) Request Sizes (CDF)**

Figure 6. Per-Method Request Size



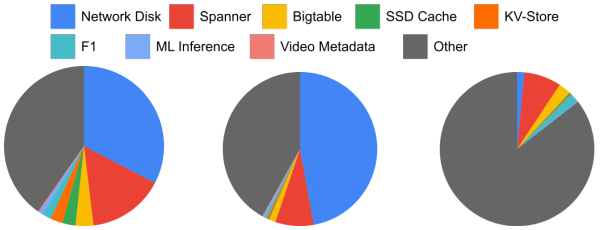
(a) Response/Request Size Ratio **(b) Response/Request Size Ratio (CDF)**

Figure 7. Per-Method Response/Request Size Ratio

Figure 6 plots the RPC request size distribution in bytes for the 10K methods. The figure shows that most RPCs are small — with the smallest a single cache line (64 B). The smallest 10% have median requests and responses under 2030 B and 188-B, respectively. Half of the methods have median requests under 1530 B, with responses under 315 B.

Although most RPCs are small, most methods have a large heavy tail. For example, P90 request and response sizes are 11.8 KB and 10 KB, respectively; P99 requests and responses are 196 KB and 563 KB — an order-of-magnitude increase over the median. This finding can predict the effectiveness of accelerators that have a maximum message size. For example, an on-NIC deserialization offload such as Zerializer [74], which can only process messages contained in a single MTU, would be able to accelerate the majority of RPCs but would miss the tail.

Although there is no previous study that characterizes the sizes of RPCs in general, our findings can be compared against other more directed studies of KV-Stores and datacenter networks. The RPC sizes in Figure 6 are similar to the distributions reported by prior studies. For example there are two studies of KV-Stores from Meta [9, 16]. Both of these studies show similar trends, where most of the transfers range from 100 B to 100 KB. There is wide range of median values across methods in Figure 6, and the Meta distributions for



(a) By number of RPC invocations. (b) By bytes transferred. (c) By CPU cycles used

Figure 8. Fraction of top RPC services.

KV-Stores show median values that are on the smaller side of median values shown in Figure 6. This is to be expected given that KV-Stores typically persist small amounts of state for otherwise stateless applications [5, 12].

When compared against studies of network flows that identify elephants and mice [6, 7, 11], we similarly find that there are a few RPC elephants and many mice, although the RPC and flow sizes are much different. This has implications for the mappings of RPCs to flows and the scheduling of RPCs inside the network. A mouse RPC that is queued behind an elephant RPC will experience a significant increase in latency. As such, avoiding elephant head-of-line (HOL) blocking is an important component of reducing tail RPC network latency.

To understand the relationship between requests and responses, Figure 7 plots the distribution of RPC response/request ratio for each of the 10K methods. A ratio of greater than 1 indicates that an RPC was read-dominant, e.g., an RPC that reads data or performs a computation that expands data. Write-dominant RPCs have a ratio lower than 1, and this includes RPCs that write or aggregate data. This figure shows that most RPC methods service both write- and read-dominant RPCs with there being a heavy tail of both large responses and large requests for all RPC methods. However, the majority of RPCs for most methods are writes because the median ratio for most methods is below 1. This finding implies that most RPC methods should expect both ingress and egress dataflow. Interference between write- and read-dominant RPCs could potentially be a problem, and this motivates future work on providing knowledge about expected request and response sizes to an RPC scheduler.

2.6 Storage RPCs are Important

Here we categorize the fleet-wide RPCs into application services. This categorization provides insights into services that consume the most resources. Figure 8 shows the fraction of RPC services by the number of RPC invocations, number of bytes transferred, and number of CPU cycles consumed by each service. As this figure shows, the top 8 applications in terms of method popularity account for 60% of total invocations. The single most popular application is Network Disk, which receives both the most RPCs and transfers the most bytes. The next most popular are Spanner, KV-Store, and F1. In addition, Figure 8b shows that the distribution of bytes

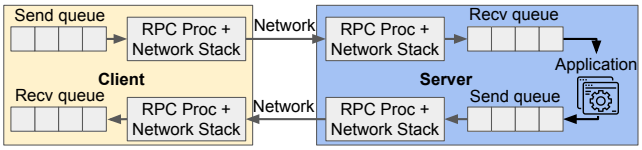


Figure 9. Components in RPC latency breakdown.

transferred differs significantly from the number of RPC calls. The Network Disk sends and receives proportionally more bytes than other applications, while the analytics services transfer fewer bytes than the other most popular services.

On the other hand, while those storage services consume a significant number of fleet CPU cycles, they proportionally use fewer cycles per RPC call compared to other applications. For example, Network Disk, which is the most popular low-latency RPC service (35% of RPCs), disproportionately utilizes less than 2% of the fleetwide CPU cycles. Longer-latency RPCs, e.g., ML Inference and F1 in our study, consume 0.89% and 1.8% CPU cycles but contribute to only 0.17% and 1.8% of RPC invocations, respectively.

These findings motivate application-specific optimizations, especially on storage systems, as storage is by far the largest distributed application in the fleet.

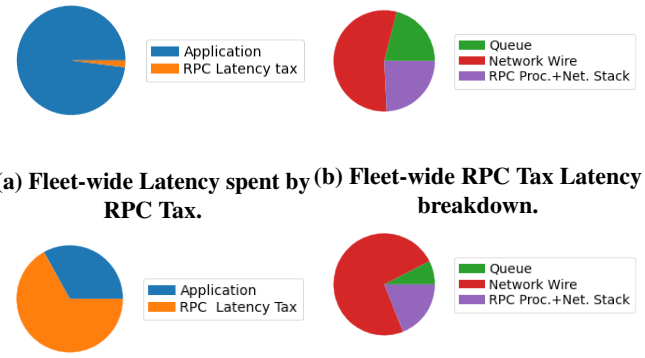
3 RPC Latency

The previous section showed that the RPC completion time varies significantly, ranging from hundreds of microseconds to hundreds of milliseconds. Therefore, optimizing an RPC requires an understanding of the RPC's components and their latencies. This section provides a fleet-wide breakdown of RPC component latencies and then analyzes individual RPC bottlenecks in eight popular cloud services.

3.1 RPC Components

We measured the latencies of different RPC components with Dapper. Although different RPC stacks may have different structures, Figure 9 illustrates the major components of the RPC stack that we measured, which are described in more detail below.

- **Client Send Queue:** Client-side RPC code places requests in a queue where they wait for transmission when local CPU and network resources are available.
- **Request RPC Processing and Network Stack:** This includes the processing and serialization latency for the RPC packets, i.e., the latency taken for marshalling and sending multiple packets, as well as the time needed for message encryption and compression.
- **Request Network Wire:** RPC request propagation latency, including wire and queuing delay in the network.
- **Server Receive Queue:** When the server receives an RPC request, it places it in a request queue, where a server thread eventually removes and processes it. Latency for



(a) Fleet-wide Latency spent by RPC Tax. **(b) Fleet-wide RPC Tax Latency breakdown.**

(c) Latency spent by RPC Tax in P95 tail **(d) RPC Latency Tax breakdown of RPC with P95 tail latency.**

Figure 10. RPC Latency Tax

this stage includes the costs of decrypting and parsing the request message.

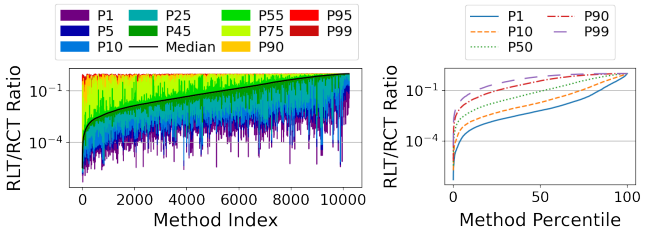
- **Server Application:** The server thread dequeues an RPC from the request queue and executes the handler for the appropriate method. If an RPC method calls another method, this latency includes the time for the subsequent calls to complete.
- **Server Send Queue:** When the application completes, the RPC system places its response in a send queue, where it waits until the network is available for transmission.
- **Response Network Wire:** Response propagation latency, which includes wire and queuing delay in the network.
- **Response RPC Processing and Network Stack:** This step includes the processing and serialization latency for the RPC responses.
- **Client Receive Queue:** When the response arrives back at the client it is placed in a response queue for processing, where it may wait if the client is busy.

With the exception of the application processing time, all other components are a result of using RPCs to access a remote service. We therefore describe these non-application latencies as the *RPC latency tax* (or more simply, the RPC tax).

3.2 Fleet-Wide Latency Variation

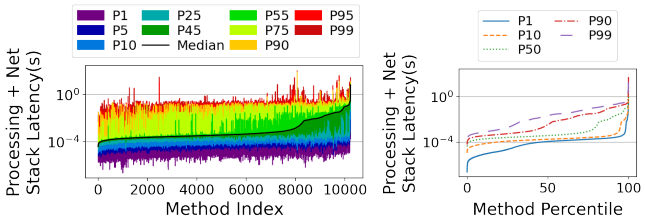
We describe how the major components contribute to the RPC completion time. In particular, we show the fraction of RPC time spent on the latency tax, and then break down the tax into RPC processing and network stack, network wire, and queuing components.

Application-processing time dominates but RPC tax can be significant. Figure 10a gives an overview of the average RPC latency tax across all RPCs. Overall, the average tax is



(a) Distribution Heatmap **(b) CDF**

Figure 11. Per-Method Ratio of the RPC Latency Tax (RLT) to RPC Completion Time (RCT).

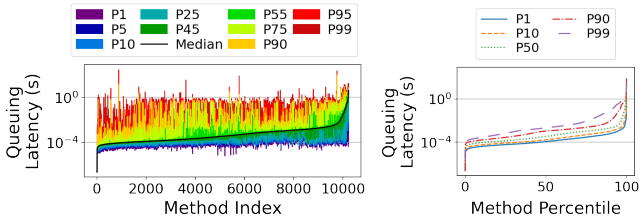


(a) Distribution Heatmap **(b) CDF**

Figure 12. Per-Method RPC Latency Distribution for Network Wire (RW) and RPC Processing and Network Stack (RN).

only 2.0% of the total completion time. Of that 2.0%, Figure 10b shows that the network accounts for roughly half (1.1% of total time), while the RPC processing and network stack component and the queuing component each contribute nearly one quarter (0.49%, and 0.43% of total time, respectively). However, for RPCs with P95-tail latency, the tax component is significant, as shown in Figures 10c and 10d. Here the distribution skews toward network-induced delay, suggesting that network congestion and/or global distribution (i.e., speed-of-light propagation delays) may be limiting tail performance.

To better understand how much of completion time is due to the RPC latency tax, Figure 11 plots the RPC latency tax ratio distributions for all RPC methods; the tax ratio is the fraction of RPC service time for which the tax is responsible. Most RPCs are bottlenecked by application-processing time; for the RPC method with the median ratio, the tax makes up only 8.6% of the total completion time. As noted before, though, the RPC tax is more significant at the tail. For the 10% of methods with the highest overheads, the median RPC tax is 38%, while the 90th-percentile RPC tax is 96%. The 99th-percentile RPC tax ratio for the top and bottom 1% of methods ranges from 0.5% to 99.99% with a median of 66%. We conclude that optimizing RPC latency requires a two-pronged approach. Optimizing server processing time is extremely important to reduce the completion time of most RPCs. At the tail, however, most method types have RPC invocations where latency comes almost entirely from the RPC tax.



(a) Distribution Heatmap (b) CDF

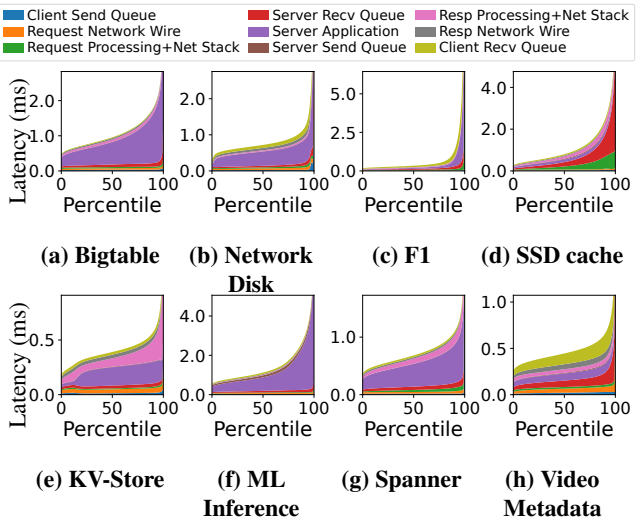
Figure 13. Per-Method Queuing Latency.

RPC tax breakdown. To quantify the relative contributions of the different RPC tax components, Figure 12 shows a per-method breakdown of the latency distribution for combined Network Wire (RW) and RPC Processing and Network Stack (RN). Figure 13 shows the per-method queuing latencies. We expect tail network latencies near the longest round-trip time across the WAN between datacenters, which is about 200 ms in our fleet. As Figure 12 shows, tail network latencies can be around this expected latency for many methods. The P99 latency for the fastest 50% of methods in network latency is 115 ms or less. There are some methods that have much lower tail latencies, with the fastest 1% and 10% of methods having a P99 latency of 6 and 19 ms, respectively; this implies that some methods avoid the cost of geographic distribution in most cases. However, at the tail, combined RPC processing and networking stack latencies are high. The slowest 10% of methods have a P99 latency of at least 271 ms. Finally, the slowest 1% have a P99 latency of 826 ms, which is significantly higher than the longest network propagation delays, suggesting that there is room for improvement in RPC processing and network performance.

Queuing latency is also a significant contributor to the RPC tax. Figure 13 shows that queuing latency is high at the tail, although on the whole it is comparable to the combined RPC processing and network latency. Half of the methods have median and P99 queuing latencies under 360 μs and 102 ms, respectively, compared to median and P99 latencies under 398 μs and 115 ms for combined RPC processing and network stack. However, for the 10% of methods that experience the highest queuing latency, median and P99 latencies are 1.1 ms and 611 ms, respectively. Thus, for many methods, tail queuing latency is much worse than median queuing latency. This implies that it may be possible to reduce tail latency for many methods by improving tail queuing with better scheduling and load balancing.

3.3 Service-Specific Latency Variation

Our fleet-wide analysis finds that there can be significant variance across different RPC services (*i.e.*, methods); here we perform an in-depth analysis of some important RPC services, most of which are leaf RPCs. In particular, we select representative RPC methods from eight production systems, listed in Table 1. These services can be categorized into three



(a) Bigtable (b) Network Disk (c) F1 (d) SSD cache

(e) KV-Store (f) ML Inference (g) Spanner (h) Video Metadata

Figure 14. CDF of RPC completion time breakdown.

	Client Send Queue	Request Network	Request RPC + Network stack	Server Send Queue	Server Application	Server Send Queue	Resp RPC + Network Stack	Resp Network	Client Recv Queue
Network Disk	0.65	0.35	1.90	4.13	22.43	0.11	0.89	0.57	5.02
SSD cache	0.78	0.18	11.30	33.58	1.64	1.98	4.13	0.00	1.59
F1	0.01	0.00	1.18	13.98	21.33	0.01	1.81	0.29	28.56
BigQuery	0.13	0.48	0.02	3.34	36.34	0.13	2.87	0.48	0.17
KV-Store	1.06	1.67	0.04	2.18	4.55	0.11	15.47	0.95	2.96
ML Interference	0.09	0.02	0.25	0.43	67.97	1.08	0.46	0.00	0.33
Spanner	0.03	0.63	1.61	2.95	26.56	0.41	3.51	0.35	0.84
Video Metadata	0.86	2.23	0.97	17.35	2.73	0.22	5.56	2.63	18.01

Figure 15. Percent improvement of tail latency (P95) with a what-if analysis.

types: storage applications, which include Bigtable, Network Disk, SSD cache, Video Metadata and Spanner; compute-intensive applications, which include F1 and ML inference; and a latency-sensitive in-memory cache KV-Store. We evaluate the distribution of latency breakdowns for each RPC across several dimensions.

3.3.1 Latency Variation Within a Cluster.

We first study the latency variation of intra-cluster RPC calls to our 8 services. For each service, we included only RPC calls (1) to the method shown for that service in Table 1, and (2) from clients located in the same cluster and datacenter as the server. Figure 14 shows the CDF of RPC latency for the selected service methods. The colors in the graphs show the breakdown of the RPC latencies into the nine latency components described in Section 3.1.

As the graphs show, not all RPCs with the same total latency have the same per-component latencies. Overall, most workloads have one dominant latency component. Based on the dominant component, we categorize the eight RPCs as *application-processing-heavy* (Bigtable, Network Disk, F1, ML Inference, and Spanner), *queuing-heavy* (SSD cache and

Category	Server	Client	RPC Size	Method Description
Storage	Bigtable	KV-Store	1 kB	Search value
	Network Disk	Bigtable	32 kB	Read from SSD
	SSD cache	BigQuery	400 B	Look up streaming data
	Video Metadata	Video Search	32 kB	Get metadata
	Spanner	Network information service	800 B	Read rows
Compute-intensive	F1	F1	75 B	Process data packet
	ML Inference	ML Client	512 B	Perform inference
Latency-sensitive	KV-Store	Recommendation service	128 B	Search value

Table 1. RPC services in this study

Video Metadata), and *RPC-stack-heavy* (KV-Store). The dominant latency components take 25–66% of the total latency at the median but increase to 30–83% at P95. However, tail latencies are significantly higher than the median: the P95 latency is 1.86–10.6× higher than the median. F1 has the largest difference, primarily because the database executes queries of varying complexity using the same RPC method.

3.3.2 Component Impact on Tail Latency. To better understand how each component impacts tail latency, we perform a "what-if" analysis by replacing each latency component of (P95) tail RPCs with its median value, one-by-one. Figure 15 shows the percentage of tail RPCs that become non-tail (i.e., move *below* the prior P95 latency) when the corresponding latency component is reduced to its median. We find that the impact is largely consistent with the categorization, i.e., the latency component that dominates the RPC latency in general is also the main cause of tail latency.

The key findings of this experiment are that, not surprisingly, the major cause of tail RPC latency in the datacenter differs among RPCs and types of applications. Prior work has focused on reducing the latency of the computation of the RPC stack [41] or the network [31] to improve RPC times. In contrast, these measurements demonstrate the importance of other RPC components, such as application service time and client/server queuing. As a result, reducing RPC latency will likely require an application-specific approach, both to choose the component to optimize and often to reduce execution time of the application methods, which dominate RPC tax mechanisms for some important applications.

3.3.3 Service Latency of Different Clusters. We now show how the latencies for each service can vary within different clusters in the cloud. To do this, for each service, we examine RPC data from dozens of different clusters and datacenters. As in Section 3.3.1, in all cases the client and server are within the same cluster, and we ensure that all RPCs are based on the same underlying hardware/software system platform.

Figure 16 shows the latency breakdown of P95-tail latencies for RPCs for each of the 8 services across different clusters. The x axis represents unique clusters in which the

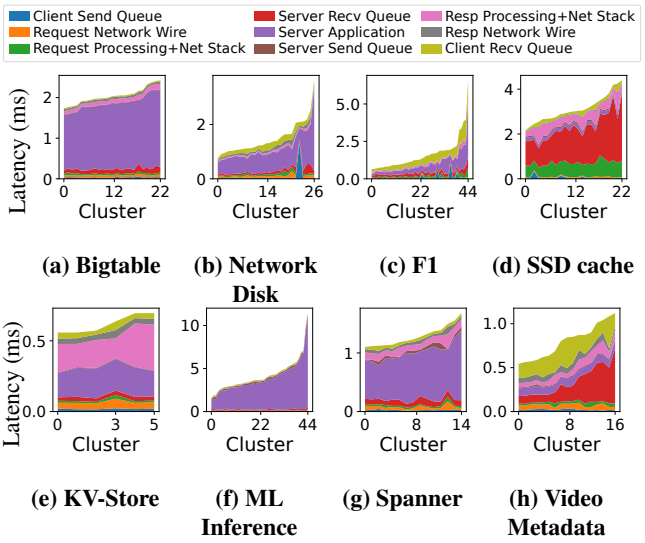


Figure 16. Distribution of latency components across clusters.

selected RPCs execute; different services run on different numbers of clusters, and in these figures the results are sorted by median latency for between 5 and 44 clusters.

The dominant component of RPC latency remains largely the same across different clusters,

yet the latency varies significantly among different clusters, with the difference ranging from 1.24 to 10×. As the system platform, RPC methods, and RPC sizes are all the same, this experiment indicates that the state of the cluster is the major cause of the differences. We refer to the system-level variables that capture this cluster state as *exogenous variables*.

3.3.4 Exogenous Variables Affecting Latency Variation. Figure 17 demonstrates the relationship between the value of these exogenous variables (x axis) and RPC latency breakdown (y axis). We pick three applications (one from each category) and four exogenous variables that have the highest variations (Table 2). Because network latency is fairly stable across different clusters, we focus on exogenous variables that capture server state.

Similar to before, RPCs with the same exogenous variable may have different component latencies, so this figure shows the average of all of the RPCs with equal exogenous variable

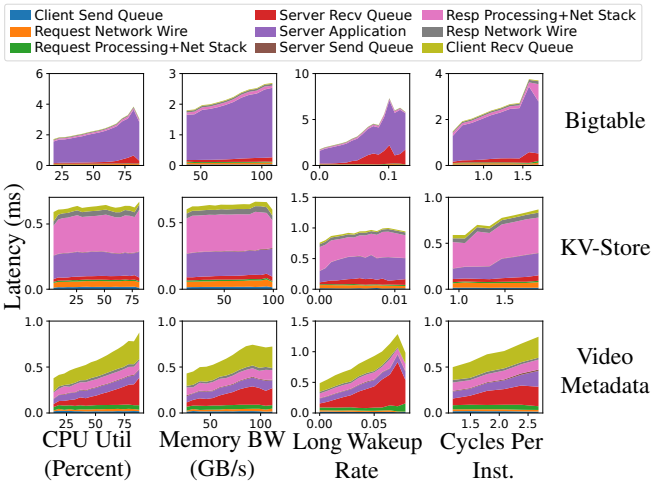


Figure 17. Relation between exogenous variable and latency components.

values. Specifically, we collected samples of exogenous variables and RPC latency and aggregated them over 30 minutes. Then we bucket RPC latency samples according to the exogenous variables (x axis). For each bucket, we select RPC latency samples with total latency near P95 (+/-1%) and plot the per-component average (y axis).

Each application category reacts differently towards these exogenous variables. Bigtable is a server-processing-heavy workload, and its performance is highly dependent on CPU utilization, memory bandwidth, wake-up time, and cycles per instruction. Video Metadata is queuing heavy, which follows a similar trend. In comparison, KV-Store, an RPC-stack-heavy workload, is most impacted by variation in cycles per instruction. We note, however, that KV-Store runs on reserved cores in our fleet, which may partially explain the lack of correlation with overall server CPU and memory bandwidth utilization. Additionally, among the applications we studied, branch misprediction and LLC miss rates are not correlated with RPC latency and their result are not shown here.

To confirm our hypothesis that exogenous variables correlate with RPC latency, we further monitor the (P95) tail RPC latency together with the value of exogenous variables of Bigtable over a 24-hour period. Figure 18 plots these values in representative fast and slow clusters. RPC latency fluctuates following the same trend as most exogenous variables, which confirms our previous findings. For example, in both the fast and slow clusters, CPU and memory bandwidth utilization both show similar trends as RPC latency. We conclude that system-level optimizations, including both the hardware platform and low-level OS details like scheduling, may benefit from application specificity. We expect future work to explore cross-layer designs that are specialized for not only different applications but for different RPCs.

Variable	Description
CPU util	% CPU utilized
Memory BW	Total memory bandwidth utilized (GB/s)
Long wakeup rate	Fraction of scheduling events longer than 50 µs
Cycles per Inst.	CPU's cycles per instruction

Table 2. Exogenous variables

3.3.5 Latency of Cross-Cluster RPCs. For some RPCs, the client and server are frequently located in different clusters. To study the impact of traversing the WAN, Figure 19 shows the median latency of an RPC to Spanner servers located in 21 different clusters. This demonstrates that when the client and server are within the same cluster or are in clusters that are close geographically, the latency is low and follow the same trend as same-cluster breakdowns. As the distance between client and server increases, the network component begins to dominate the RPC latency.

Unfortunately, most of this latency is unavoidable as the network latency is bounded by the speed of light. We cross-validated the cross-cluster latency in Figure 19 and found that the latency closely matches the actual wire latency. Therefore wire latency, not congestion, contributes to the majority of the network latency of the average RPC.

We conclude that, *on average*, the room for network latency optimization in a global cloud environment is limited as some communication latency is unavoidable. However, one of the main reasons for cross-cluster RPC is a lack of data locality, i.e., RPC servers are not located close to the data being processed. As such, it is critical to optimize data locality in large-scale distributed RPC systems.

4 Resource Utilization of RPCs

This section studies the CPU costs of RPCs, which we refer to as the RPC *cycle tax*. We also examine the effectiveness of RPC load balancing. Understanding these costs can help future research make RPCs more efficient.

4.1 CPU Cycle Breakdown

Figure 20 shows the RPC cycle tax across the entire fleet with respect to CPU cycles consumed: roughly 7.1% of all cycles. Further, the right-hand pie chart shows that there are many different components that contribute a significant fraction of the cycles. The single biggest consumer of CPU cycles is compression, at 3.1% of all cycles. The next-most-significant consumers of CPU cycles are networking and serialization, at 1.7% and 1.2% of all total cycles, motivating research on serialization offload [56, 74].

4.2 Fleet-Wide CPU Cycle Variation

Figure 21 shows a per-method breakdown of RPC costs. In this figure, costs are measured in terms of normalized CPU cycles, a unit that reflects the varying performance across

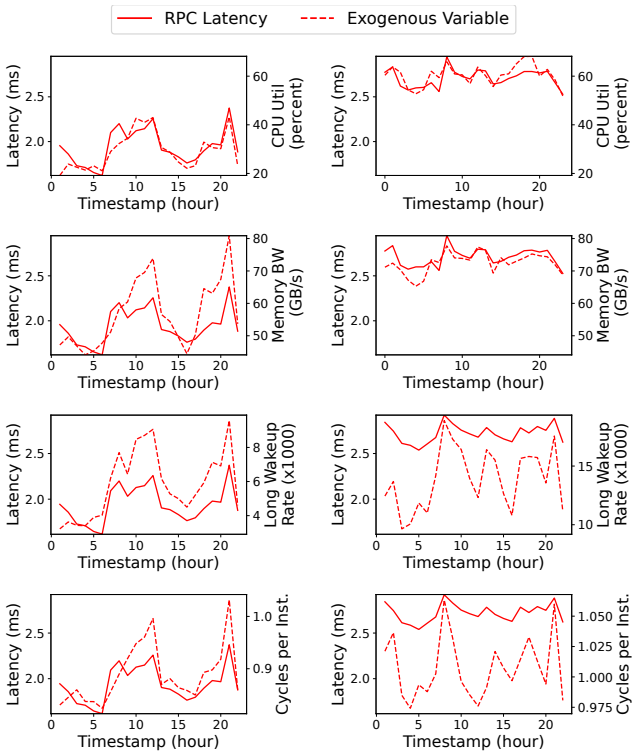


Figure 18. Comparison of exogenous variable and latency between different clusters (Bigtable)

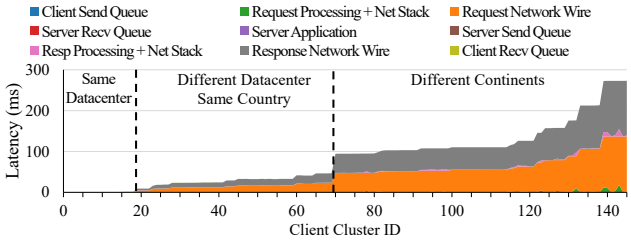
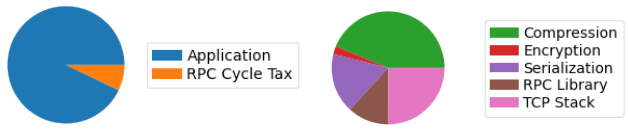


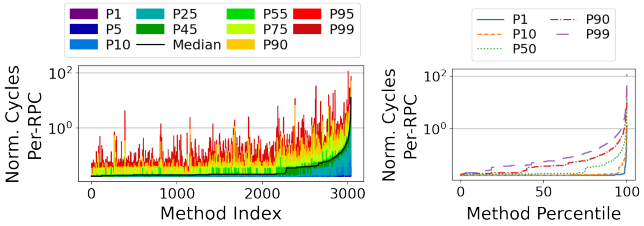
Figure 19. Spanner Cross-cluster latency breakdown.



(a) Total cycles consumed by the RPC cycle tax. **(b) RPC cycle tax breakdown.**

Figure 20. RPC Cycle Tax.

different CPU architectures and generations present in our fleet. Not all RPC samples collected by our tracing service are annotated with CPU cost information, so this figure has fewer methods than our previous analyses.



(a) Per-Method CPU Usage Distribution. **(b) CDF**

Figure 21. Per-Method RPC CPU Cycles.

Similar to our previous per-method analysis, this figure shows that RPC CPU utilization is heavy tailed, with less difference between the minimum and maximum values on a per-method basis. For example, the cheapest 10% of RPC calls only change from 0.017 normalized cycles or less to 0.02 normalized cycles or less when moving from the cheapest-10% to cheapest-90% of methods. However, in contrast, the most-expensive 10% of calls span 0.02–0.16 normalized cycles or more between the cheapest 10% and 90% of methods.

When RPC calls are cheap, there tends to be low variance: the difference between the P1 and P99 throughput for the cheapest 1% of methods is within a factor of two. In contrast, almost all other methods have a heavy tail where the P99 RPC costs are one-to-two orders of magnitude more than the median; there are no methods that have high CPU overheads with low variance. This high variation has significant consequences for RPC scheduling, load-balancing, and queuing [27, 40, 53]. If RPC processing times are not known in advance — which is not always possible because processing-time prediction is a hard problem in general [28] — then this heavy-tailed cost distribution is likely to lead to significant HOL-blocking latency. If an RPC with low CPU cost unluckily ends up queued at a server that is currently processing an expensive query, then it could see significant latency inflation. This spread also implies that any load balancing algorithm that treats different RPCs as being equal is likely going to lead to significant CPU imbalance. Further, improving load balancing is a challenging problem because it is difficult to know in advance which RPCs will be expensive. For example, we found that neither RPC size nor RPC latency is correlated with RPC CPU utilization.

4.3 Load-Balancing Resources

RPC load balancing determines how load is distributed across servers. Figure 22 shows a CDF of the ratio between used CPU resources and the allocated CPU resource limit across all clusters (solid lines) and across different machines in the same cluster (dashed lines) for each application. We observe that load is significantly imbalanced across clusters. Our load balancer considers network latency when distributing RPCs among remote clusters, and balancing server CPU load across clusters is not an explicit goal. That said, avoiding overload at

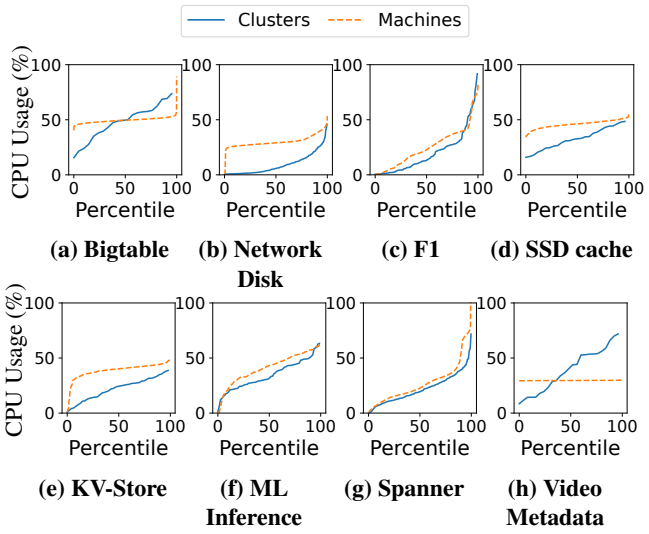


Figure 22. Distribution of CPU usage across different clusters and different machines in the same cluster.

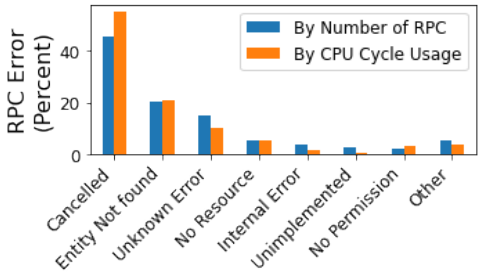


Figure 23. The relative percentage of different RPC error types.

any particular cluster is, and we find that, for some services, tail utilization can approach the limit. This motivates finding new ways to better balance load across cells while still ensuring network latency is low.

Conversely, the dashed line in Figure 22 shows the CDF of the same ratio but across different servers in the same cluster. Overall, the load among servers has a much smaller variation, except for Spanner, F1, and ML Inference. These applications have some servers that are nearly fully utilized, suggesting improvements are needed in intra-cluster load balancing as well. However, this is a hard problem because load balancing in some of the applications are data dependent and may suffer from limited parallelism.

4.4 RPC Cancellations and Errors

RPCs are not guaranteed to complete. We find that 1.9% of all of the RPCs issued during our period of study resulted in errors. There are a large variety of reasons that RPCs may experience an error, but all RPC errors waste resources.

To better understand the sources and costs of RPC errors, Figure 23 plots the contribution of different error types to the total in terms of both percentage of errors and wasted CPU cycles. "Cancelled" is the most-common type of error, which constitutes 45% in number and 55% of CPU cycles. We suspect request hedging [23] is responsible for most cancellations in a deliberate attempt to reduce tail latency. Moreover, we observe that while many error types have relatively similar contributions to frequency and CPU cost, cancellations consume an out-sized fraction of CPU cycles, making them more expensive than most. Although it is unclear exactly what fraction of RPCs are canceled due to other reasons like a user cancelling a job or query, this finding suggests there is value in further understanding the overheads and trade-offs associated with request hedging.

Unlike cancellations, which may be side effects of a tail-latency-reduction strategy, other classes of errors are expected to increase tail latency. As such, we believe that there is a need for future research on the topic of reducing the relative number of RPCs that experience errors in large-scale distributed systems. The second-most common source of error is "entity not found", which comprises 20% of total RPC errors and consumes 21% of wasted CPU cycles due to RPC errors. It is one of the error types that can possibly be eliminated or significantly reduced by preventing requests to unavailable entities. Other errors such as "no resource" and "no permission" can also be possibly mitigated.

5 Implications

This section briefly highlights the implications of our results based on the data we presented. It presents our key insights about RPC behavior and problems with today's RPC stacks, and it highlights potential opportunities for improvements and optimizations at the software and hardware layers.

5.1 RPC Behavior and Problems

This work provides a comprehensive study of the fleet-wide RPC characteristics and behaviors of Google's internal applications. Our findings have important implications with regards to RPC behavior and problems that can be addressed.

Millisecond, not just microsecond timescales. Many of the RPCs in our cloud are on the scale of milliseconds, not microseconds. Reducing CPU utilization is in many cases more critical than saving tens of microseconds of latency. Yet many prior proposals focused only on microsecond-scale latency improvements [38, 41, 53, 55].

Queuing matters. Our study (Fig. 11 and 12) shows that queuing is a major contributor to tail latency. Prior work has focused on reducing network tail latency [31, 51, 54], which only solves part of the latency problem in RPC systems.

Congestion still impacts the WAN. We find that tail RPC network latencies are much larger than the maximum median inter-datacenter latency. Although prior work on private WANs suggests that congestion is a solved issue [36, 39, 45, 75], we show that network latency from congestion has a significant impact on the tail network throughput of RPCs.

Tail latency is a bigger problem than average latency. While the RPC latency tax is only 2% on average, it can climb to 96% at 90th-percentile. Therefore, there is still a need to reduce the RPC latency tax at the tail. New RPC optimizations could provide predictable performance and potentially have a significant impact.

RPC errors. Our report of real-world errors and their distributions shows that (a) RPC hedging is costly (55% of wasted CPU cycles), and (b) unavailability of RPCs and other resource and permission issues account for 20% of wasted RPCs. This suggests future research on mitigations, e.g., service availability prediction.

5.2 Software Optimizations

There are different software components that impact RPC processing. Our findings have implications on how these software components could be changed to reduce latency or CPU cycles utilization.

Improved scheduling and placement. Queuing latency consumes 21% of the RPC latency tax, so better scheduling is likely to reduce RPC completion times. Latency suffers when clients and servers are not co-located. As a result, adding support to a cluster manager for co-locating RPCs from the same RPC tree could significantly reduce latency.

Load balancing. One of the major sources of latency variation comes from system balance and congestion issues. For example, if the system exhibits high server and memory bandwidth utilization or saturation, then tail latency can increase significantly, particularly for RPCs that are bottlenecked by server processing and queueing latency. We expect that a cross-layer load-balancing mechanism that takes into account both RPC type information and system resource states will greatly reduce the latency variation.

Method-specific software optimizations. As previously noted, the 10 most popular RPC methods account for 58% of all calls and the top-100 account for 91% of all calls. Therefore, a small number of targeted method-specific optimizations could potentially have a significant impact on a large fraction of RPCs. Our service-based latency analysis shows that these optimizations must address the main bottlenecks in each service. For example, compute intensive services are bottlenecked by their processing time (e.g, ML inference), while light-load services are limited by queuing delay (e.g., video metadata indexing). Prior work mostly focused on optimizing light-load services [41, 71], but in reality, future research in service-specific optimizations targeting each of their main bottlenecks will be crucial as well.

5.3 Hardware Optimizations

Hardware accelerators have emerged as a promising approach to scaling the performance of datacenter applications in the face of Dennard scaling limitations. This is because hardware acceleration can potentially reduce latency, energy consumption, and total costs.

Optimizing common operations. A number of common compute or data-intensive operations, such as compression, encryption and serialization, are used in RPC as well as in the network stack. Prior work [38, 41, 53, 55] has evaluated optimizations, but without including these operations, which are required in the cloud environment. On the other hand, hardware accelerators for these operations are common on many systems or NICs and could be included in RPC processing.

RPC Library acceleration. Figure 20a shows that the RPC Library only takes a small fraction of total CPU cycles (1.1%). Therefore, accelerating the RPC Library using a SmartNIC/xPU may not provide the highest value when compared to other common data center tax operations (e.g., serialization/deserialization and compression).

Storage dataflow accelerators. The majority of our RPC invocations and data transferred are from two applications (Figure 8) — Network Disk and Spanner — which are data intensive. This demonstrates the potential for accelerating data movement [20, 57].

Method-specific hardware optimizations. Accelerators must cover a significant fraction of the CPU cycles consumed by the fleet to provide cost-efficiency benefits [20, 43, 57]. As most CPU cycles are consumed by a few services (the top 8 services account for 60% of all RPC calls), a few method-specific accelerators could potentially have a significant impact on the fleet.

5.4 Limitations

We briefly note two limitations to this study. First, it analyzes RPCs from a single hyperscaler. Others may have different structures that might lead to different results. However, we believe that it is likely that many of our findings generalize, and that RPCs in cloud applications at other companies are likely to share similar properties and behavior, given the similarity of services that clouds provide. Our study provides a basis for understanding this workload and comparing with later studies. Conducting a cross-cloud study at this level and magnitude would obviously be challenging.

Second, this study focuses primarily on RPCs sent over TCP. RDMA has become increasingly important recently as an alternative transport to TCP for RPCs [19, 25, 34, 41, 50, 77]. However, we have focused on RPC over TCP and leave the study of RDMA to future work.

6 Related Work

There are two general classes of related work: previous studies of datacenters and research on improving RPCs.

Generally, this paper is complementary to previous datacenter studies because it adds detailed data and analysis of RPCs in the modern cloud environment, which was not previously available. This includes prior studies of the characteristics of network traffic in datacenters by Roy *et al.* [60], Alizadeh *et al.* [7], and Benson *et al.* [11]. In this prior work, the TCP

flow is the base unit. This paper providers further insight into the behavior of these TCP flows in datacenters that are sending and receiving RPCs. For example, these prior studies have found that TCP flows exhibit on/off traffic patterns, and this can be in part caused by the heavy tailed RPC size distributions that we observed.

In addition to network studies, there are also datacenter studies. Kanev *et al.* [42] perform CPU profiling of Google's datacenters, and Gonzalez *et al.* [32] studied hyperscale big data processing at Google. These studies look deeper into the CPU behavior of Google's internal applications, while this paper looks into the RPC behavior that is generating the CPU load for these applications. Further, these other datacenter studies reach similar conclusions about the potential benefits of using accelerators for important applications.

Sriraman *et al.* [69] profile Meta's microservices for acceleration opportunities. Our RPC analysis complements these studies by focusing on RPC services within the workloads and providing a detailed analysis of non-application overheads from the RPC latency tax. Luo *et al.* [48] characterize microservice dependencies and performance. They similarly find that microservice call graphs are heavy tailed. ServiceRouter is a global service mesh used to route RPCs at Meta [62], and Saokar *et al.* [62] found that the cloud-scale applications that use ServiceRouter show similar trends to those used at Google. Huye *et al.* [37] study the microservice topologies and workflows of a few of Meta's internal applications. Our study builds upon this study by analyzing more RPC methods and by performing a more in-depth latency analysis.

Our study is also closely related to efforts to improve RPC performance. For example, Chen *et al.* argue that RPCs should be an OS-managed service [18]. This paper provides insights into the expected benefits of such an approach.

Wang *et al.* argue that it is time to add distributed memory to RPCs [73]. The RPC Chain is a new abstraction that can reduce network latency by chaining multiple RPC invocations [67]. This paper helps further motivate these research directions by showing that nested RPC call trees can be deep. This is because the potential benefits of both of these systems increase with the depth of the RPC call tree.

Next, there is related work on reducing the latency and CPU overheads of RPCs. Erms is an efficient resource management system for microservices that is intended to guarantee SLAs in shared microservice environments [49]. CRISP is a tool for analyzing RPC critical paths that was used at Uber to reduce tail latency [76]. This paper motivates the need for these systems and others that can reduce tail latencies by showing that RPC latencies are high at the tail.

eRPC is a system that onloads transport protocol processing to reduce latency [41]. Our findings rebut some of the previous understanding of eRPC. As we find that most of our RPCs are millisecond-scale, this design choice seems worse in practice than using existing RDMA for transport, which can reduce the CPU overheads of messaging.

There have been many accelerators for transport and RPC stacks. For example, Zerializer [74], Raghavan *et al.* [56] and Karandikar *et al.* [43] introduce accelerators for marshalling/demarshalling RPCs. Chiosa *et al.* [20] demonstrates an accelerator for offloading compression and encryption in SAP HANA database. Tonic [8] and AccelTCP [52] are hardware accelerators for the TCP transport protocol. Dagger offloads the entire RPC stack to an FPGA-based NIC [46]. NeBuLa is a CPU architecture optimized for accelerating microsecond-scale RPCs [72]. nanoPU creates faster paths from the network to CPU [38]. We provide insight into the expected benefits of these accelerators by showing the expected percentage of CPU cycles that could be saved across the fleet.

nanoPU also discusses different application classes that could benefit from a CPU fastpath for messaging, and this includes μs-scale services. Although we found that most services at Google at not μs-scale, this does not mean that it is not possible to use μs-scale services at Google. However, decomposing existing applications into smaller services to better utilize new hardware like nanoPU is a challenging problem, and this helps motivate systems that aim to do this, like Nu [61] and ServiceWeaver [33].

Shenango is a centralized software RPC load balancer, and RingLeader [47] and Turbo [63] are hardware accelerators for RPC load balancing. Shinjuku [40] and Caladan [27] are CPU schedulers that aim to isolate short- and long-running applications. Our work motivates systems like these that can reduce RPC queuing latency, as we show that queuing latency contributes a significant fraction of the RPC latency tax.

7 Conclusions

This paper presents an analysis of RPCs in Google's global-scale cloud environment that supports its billion-user web applications. We study over 700 billion RPC samples across 10,000 distinct RPC methods, and report on the growth of RPC usage over a 700-day period. Our findings provide new insights into the characteristics and behavior of RPCs that have the potential to help shape the direction of future datacenter systems. For example, we find that RPCs operate on different timescales than assumed by prior work, and that latency overheads of RPCs are different from those assumed by prior work. We also show that RPCs are increasingly important and their growth is outpacing the growth in compute cycles in our fleet. This motivates research on reducing both the RPC latency tax and RPC cycle tax, and we present breakdowns of the components of both to light a path toward reducing them. Overall, RPC is the fundamental building block for widely distributed applications in modern computing environments. Our measurements of some of the largest-scale cloud applications help us better understand the organization and behavior of these applications and motivate the improvement and development of RPC systems in the future.

References

[1] Binder, android developer references. https://developer.android.com/reference/android/os/Binder.

[2] Lambda, 2022. https://aws.amazon.com/lambda/.

[3] Azure service fabric, 2023. https://azure.microsoft.com/en-us/products/service-fabric.

[4] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor, editors. *Monarch: Google's Planet-Scale In-Memory Time Series Database*. VLDB Endowment, 2020.

[5] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. Association for Computing Machinery, 2019.

[6] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2010.

[7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2010.

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Association for Computing Machinery, 2012.

[10] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 2003.

[11] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*. Association for Computing Machinery, 2010.

[12] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. 2016.

[13] Betsy Beyer, Niall Murphy, David Rensin, Stephen Thorne, and Kent Kawahara. *The Site Reliability Workbook*. 2018.

[14] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[15] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2020.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006.

[18] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as an os-managed service. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2023.

[19] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2019.

[20] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. Hardware acceleration of compression and encryption in SAP HANA. In *48th International Conference on Very Large Databases (VLDB)*, 2022.

[21] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets)*. Association for Computing Machinery, 2012.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J.J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and Wilson Hsieh. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2014.

[26] Sérgio Fernandes and Jorge Bernardino. What is BigQuery? In *Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS)*. Association for Computing Machinery, 2015.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[28] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. On the use of ML for blackbox system performance prediction. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021.

[29] 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. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2019.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[31] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of the 19th*

USENIX Conference on Networked Systems Design and Implementation (NSDI). USENIX Association, 2022.

[32] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2023.

[33] Google. Service weaver. https://serviceweaver.dev/.

[34] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2016.

[35] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2013.

[37] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2023.

[38] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A nanosecond network stack for datacenters. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2021.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally deployed software defined WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2013.

[40] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2019.

[41] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2019.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2015.

[43] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[44] Abhishek Kumar, Jayant Kolhe, Sanjay Ghemawat, and Louis Ryan. gRPC Protocol, July 2016. Work in Progress, https://datatracker.ietf.org/doc/draft-kumar-rtgwg-grpc-protocol/00/.

[45] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2015.

[46] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[47] Jiaxin Lin, Adney Cardoza, Tarannum Khan, , Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently offloading intra-server orchestration to NICs. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2023.

[48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2021.

[49] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with sla guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2018.

[52] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020.

[53] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2019.

[54] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2014.

[55] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. Association for Computing Machinery, 2021.

[57] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[58] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 2010.

[59] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[60] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2015.

[61] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2023.

[62] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In *Proceedings of the 17th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2023.

[63] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. Turbo: SmartNIC-enabled Dynamic Load Balancing of µs-scale RPCs. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[64] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. In *39th International Conference on Very Large Databases (VLDB)*, 2013.

[65] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[66] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[67] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2009.

[68] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013.

[69] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[70] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2014.

[71] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2017.

[72] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula RPC-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2020.

[73] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In reference to RPC: It's time to add distributed memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. Association for Computing Machinery, 2021.

[74] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. Association for Computing Machinery, 2021.

[75] Chi yao Hong, Subhasree Mandal, Mohammad A. Alfares, Min Zhu, Rich Alimi, Kondapa Naidu Bollineni, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Jeffrey Liang, Kirill Mendelev, Steve Padgett, Faro Thomas Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jon Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2018.

[76] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2022.

[77] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2015.