# Characterizing and Mitigating Overheads of Distributed Applications

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

---

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Korakit Seemakhupt

May 2025

# Acknowledgments

I am grateful to many people who have contributed to shaping this dissertation. This dissertation would not have been possible without their influence, support, advise and mentorship.

First, I would like to express my sincere gratitude to my advisor, Prof. Samira Khan, for her invaluable support throughout my journey. Samira consistently encourages me to tackle challenging problems and allows me the freedom to explore original and unconventional ideas. Her guidance extends beyond academics, as she generously advises me on career decisions.

Second, I would like to thank many of my co-authors, including David E Culler, Arvind Krishnamurthy, Henry M Levy, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz ,Alex C Snoeren, Brent E Stephens, Hassan Wassel and Soheil Hassas Yeganeh; without their support, none of my work would be possible.

Third, I would like to thank my colleagues, including Divya Bagla, Amel Fatima, Beenish Gul, Yasunari Kato, Khyati Kiyawat, Sihang Liu, Suyash Mahar, Alenkruth Krishnan Murali, Yasas Senevirathne and Akhil Shekar; for their continuous motivational support, enriching discussions, and collaborative spirit. The insightful conversations we've shared have sparked numerous ideas, laying the foundation for much of my research. Their willingness to exchange perspectives and provide constructive feedback has been instrumental in shaping my thought process.

I would also like to express my gratitude to the members of my Ph.D. committee, who have graciously contributed their time and expertise to shaping this work. Their insightful suggestions and constructive criticism have significantly enhanced the quality of my research.

Lastly, I would like to extend my heartfelt gratitude to my parents. Without their hardship, pursuing this study would have been impossible.

# Abstract

Today's applications, such as social networks, data analytics, and retrieval search, rely on collaboration between devices at the user's end (edge devices) and powerful cloud systems. However, optimizing these systems remains a challenge. First, these cloud-scale systems comprise multiple large-scale distributed services whose communication characteristics are not yet well understood. Secondly, most datacenter traffic involves interactions with storage services, where remote data access includes not only storage device access and data processing, but also RPC stack and queuing latencies. Third, these applications must serve end users on edge devices such as mobile phones. The network latency between edge and cloud and the processing latency within the cloud could be unpredictable. Although local processing does not suffer from this variation, it suffers from limited computing resources such as memory capacity.

This work addresses these three critical challenges in edge-cloud systems. First, we study the overhead of communication within real-world cloud systems to understand and pinpoint potential performance bottlenecks. Secondly, we mitigate the latency of the RPC stack in remote data access in the datacenter by storing persistent data on network devices, effectively moving the latency of the server out of the critical path. Finally, we address the problem of limited resource on edge devices for running data-intensive applications by introducing a memory efficient Retrieval Augmented Generation (RAG) system that allows retrieval search of large database locally on resource-constrained edge devices.

# Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

---

Korakit Seemakhupt

This dissertation has been read and approved by the Examining Committee:

---

Samira Khan, Advisor

---

Yixin Sun, Chair

---

Seokhyun Chung

---

Chang Lou

---

Muhammad Shahbaz

---

Brent Stephens

Accepted for the School of Engineering and Applied Science:

---

Engineering Dean, Dean, School of Engineering and Applied Science

May 2025

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Modern distributed applications span an unprecedented scale and diversity – from hyperscale cloud services serving billions of users to machine learning applications running on resource-constrained edge devices. Optimizing these systems is critical directly impacts user experience through lower latency and reducing operational costs for cloud providers. Effective optimization requires a comprehensive, end-to-end understanding of these systems. However, gaining such insights poses several challenges. First, the sheer scale and diversity of these systems necessitate collecting vast amounts of data. Second, accurately capturing end-to-end system behaviors involves aggregating and integrating data from various sources, including servers running distinct services and networking components, into a cohesive, relatable data. A thorough understanding of how distributed applications interact and communicate is therefore essential for identifying optimization opportunities and enhancing overall system performance. In this study, we characterize RPCs within Google's internal cloud-scale production environment, spanning over 100 production clusters. This characterization enables us to identify critical attributes, including the number of RPCs and their growth patterns, detailed breakdowns of RPC latency components, and the structural aspects of RPC calls. By analyzing RPC behaviors in global-scale applications at one of the world's largest hyperscalers, we can precisely pinpoint application-specific bottlenecks.

One of our key findings is that most datacenter traffic involves communication with storage services. Figure 1.1 shows the fraction of storage RPC by number of invocation in one of the major cloud system. The majority of RPC (over 60 percent) are to storage service. This identifies remote data

access and storage operations as critical performance bottlenecks. Optimizing these storage operations is critical since it could significantly enhance the performance of a wide range of applications. Despite recent advancements in storage technologies such as phase-change memory (PCM) [3] that greatly reduces the latency of local data accesses. Remote data accesses continue to introduce substantial overhead due to network communication latency. Consequently, the second challenge addressed by this work is effectively utilizing these advanced storage devices while minimizing the overhead associated with remote data retrieval across networks.

Another significant finding is that applications utilizing user data, such as Retrieval-Augmented Generation (RAG) [4], inherently require accessing user-specific information. This data may reside either locally on user devices or remotely within cloud-based services. However, users mainly interact with these applications via edge devices, such as smartphones. This reliance on cloud-based processing requires transferring user data to remote servers or retrieving it from distant data stores, introducing significant network latency that adversely impacts user experience. Furthermore, transmitting user data to servers for processing can risk exposing sensitive information, raising privacy concerns. Alternatively, local processing on edge devices eliminates this network overhead but faces the limitation of restricted computational capabilities and memory resources. Figure 1.2 shows the latency breakdown of Edge and Cloud based RAG systems. Although the Cloud-based RAG system incurs network latency, it benefits from abundant cloud resources that accelerate request processing. In contrast, the Edge-based RAG system faces resource constraints, resulting in higher request processing latency. Thus, the third challenge identified in this work is how to effectively enable responsive and confidential processing of data-intensive RAG applications on resource-constrained edge devices.

**Thesis statement:** Optimizing modern distributed systems requires a comprehensive understanding of end-to-end performance, enabling targeted optimizations at critical bottlenecks.

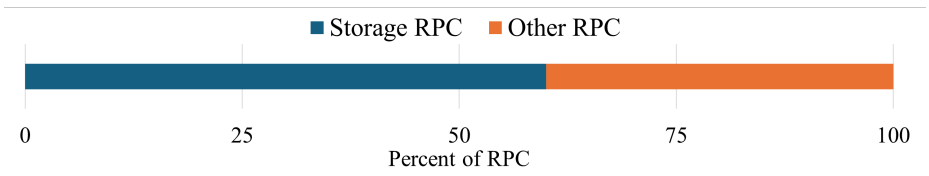



Figure 1.1: Fraction of Storage RPC in a major cloud system.

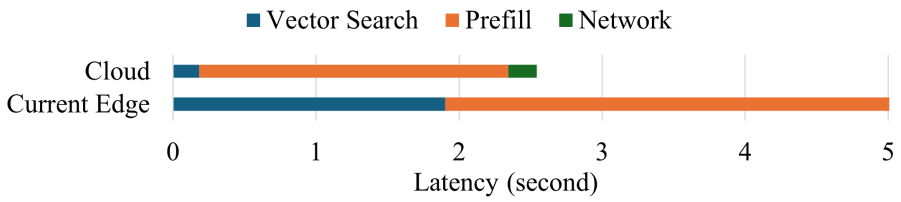



Figure 1.2: Latency breakdown of edge and cloud based RAG systems.

## 1.1 Theme 1: Characterization of Communication within Datacenter

Cloud computing has become the backbone of modern web services such as data storage, analytics and ML. Cloud services are typically broken down into smaller, independent services that talk to each other [5]. This communication usually happens through a method called Remote Procedure Call (RPC). RPC essentially allows these separate parts of a larger application to work together seamlessly, similar to a regular function call [6] within a single program while hiding additional complexities such as network stack processing, serialization/deserialization, encryption/decryption and connection management within an RPC stack. Since RPCs are the backbone of communication within cloud applications, understanding them is crucial for analyzing the behavior and performance of these large-scale distributed systems. Prior research has focused on data center network behavior [7–9] and CPU behaviour [10] within data centers, but there's a critical gap in understanding RPCs at cloud scale. This is a major limitation because RPCs essentially dictate the workload for networks, CPUs, and storage systems within the cloud.

## 1.2 Theme 2: Accelerating data durability of remote storage systems

Traditional remote storage performance suffers not only from limitations of storage devices and network hardware, but also from other factors such as wire latency and network stack latency. While traditional read caching helps with read access by storing data closer to applications (on the host or in network) [11], it doesn't improve the latency of write requests. This is because write requests require the data to be durably stored and visible to all users before the application can continue. This lock-step approach forces applications to wait through end-to-end delays, including network stack on the host, wire and switch latencies, and finally the storage system itself. To address this

bottleneck, we propose PMNet which integrates Persistent Memory, a fast and durable storage type, directly into network devices. This allows PMNet to temporarily store incoming write requests and unblocking the application immediately. The data is then durably written to the storage server in the background. PMNet's durability ensures recovery of in-flight requests caused by temporary hardware or network failures. Additionally, PMNet leverages data center network design to tolerate hardware issues by storing data across multiple network devices. PMNet can also be combined with existing in-network read caching solutions for further read access acceleration.

## 1.3 Theme 3: Optimizing Data-Intensive ML for edge systems

A recent rise of applications such Image Generators [12] and Smart Assistance [13] requires vast amount of data and processing power. Currently, these applications are limited on local devices by factors like memory, processing power, and battery life. This often forces a trade-off: simplified, less accurate models run locally, while complex tasks are offloaded to the cloud. However, these solution cannot exploit user's data in edge device without sending confidential data to the cloud to retrain the model at the cost of latency due to network limitations, precious energy and bandwidth for data transfer, and potential privacy concerns. While advancements like Retrieval Augmented Generation (RAG) allow large language models to process user data without constant training updates, the overhead of indexing data locally can still be significant. Our research explores this trade-off between local and cloud processing, focusing on ways to leverage user data on edge devices and avoid sending sensitive data to the cloud.

## 1.4 Summary

The organization of this work is as follows. Chapter 2 discusses the first theme of this work, RPC characterization, aimed at systematically analyzing and understanding the performance of remote procedure calls within large-scale distributed environments. Chapter 3 presents the second theme is optimizing network storage systems which is the most dominant application in Datacenter RPC in terms of both number and traffic volume. In Chapter 4, we introduce enabling memory efficient RAG on edge devices that mitigates the security issue of processing sensitive data on the cloud and long local processing latency. We discuss related works in Chapter 5. Finally, Chapter 6 concludes this work and its contribution.

# Chapter 2

# Characterization of Communication within Datacenter

## 2.1 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) [5, 14–18]. 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 [6], 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 [19–21].

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

[1] RDMA [22–26] systems are increasingly used along with RPC for data movement, but we focus here on RPC.

performance of modern cloud-based widely-distributed applications. While prior work has analyzed the behavior of datacenter networks [7–9] and the microarchitectural CPU behavior of datacenter applications [10], 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 [27].

This work 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 work 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.

This chapter is organized as follows. Section 2.3 explores the scope and complexity of RPCs in Google's hyperscale environment. In Section 2.4, we examine the RPC latency and its major components that affect RPC completion time and variance. Section 2.5 considers resource utilization in RPC services across the datacenter, focusing on the opportunity for load-balancing. We discuss implications of our research in Section 2.6.

## 2.2 Motivation

### 2.2.1 Goals

To help better shape the direction of future research on optimizations and accelerators for RPCs,

we answer the following questions in the rest of the paper:

- *Section 2.3* explores the scope and complexity of RPCs at one of the world's largest hyperscaler infrastructure companies. Many prior works on RPC optimizations assume specific characteristics, such a micro-second level latency or small RPC sizes or few nested RPC hierarchies. We analyze these characteristics to uncover the true scope of the hyperscalar RPCs.
- *Section 2.4* examines the *RPC latency tax* that hyperscalars pay to execute a service, broken down into the major component that affectse RPC completion time and its variance. How does this bottleneck vary across services and across datacenter? What optimizations will help to improve tail latency?
- *Section 2.5* considers resource utilization in RPC services across the datacenter, focusing on the tradeoff between resource utilization and latency, and the impact of *tail tax* – hedging of RPCs to avoid errors or long tails. How can we design an efficient, fast, and balanced system?

## 2.3 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 [28], BigQuery [29], Bigtable [30], F1 [31], GFS [32], and Chubby [33]). 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 [18], 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 [31, 34]. Data for these services is stored remotely in a network filesystem that replicates blocks across multiple machines in different datacenters for fault tolerance [32]. 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 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.

### 2.3.1 Methodology

In our analysis, we primarily focus on the most common RPC stacks at Google: Stubby [18] and gRPC [19]. Our analysis is performed using three different Google-internal monitoring tools, Monarch [35], Dapper [36], and GWP [37]. We particularly focus on the distribution of RPC completion time, size, and depth, as prior works propose optimizations based on these RPC properties [38–46].

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 [36]. 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.

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 $99^{\text{th}}$ percentile is well defined.

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

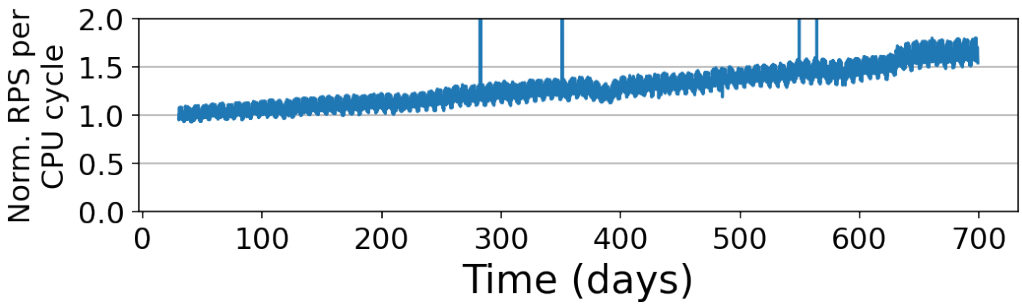



Figure 2.1: **Normalized RPS per CPU cycles consumed over time.**

We use GWP (Google-Wide Profiling) [37] 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.

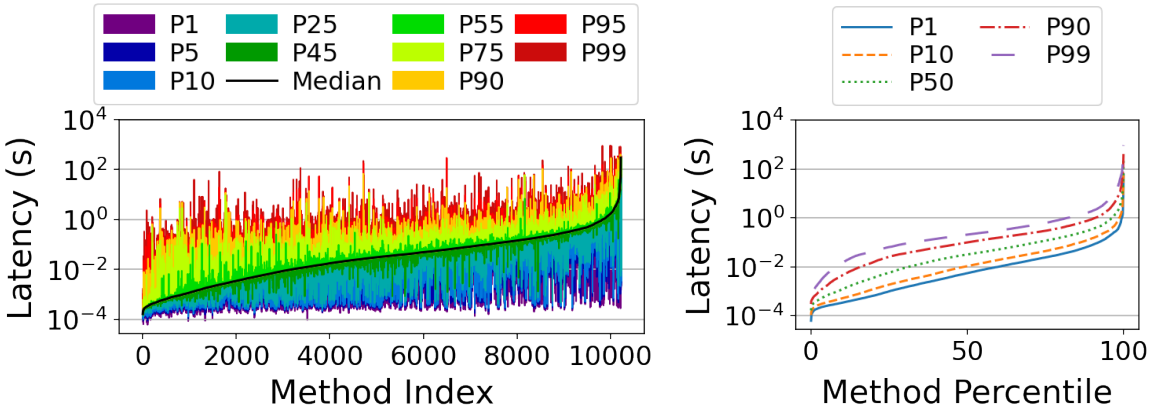
### 2.3.2 Why is RPC Evaluation Important?

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 2.1 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 2.1, 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 [47–49] is reducing the number of CPU cycles consumed per RPC, and this is likely to accelerate this trend even more in the future.

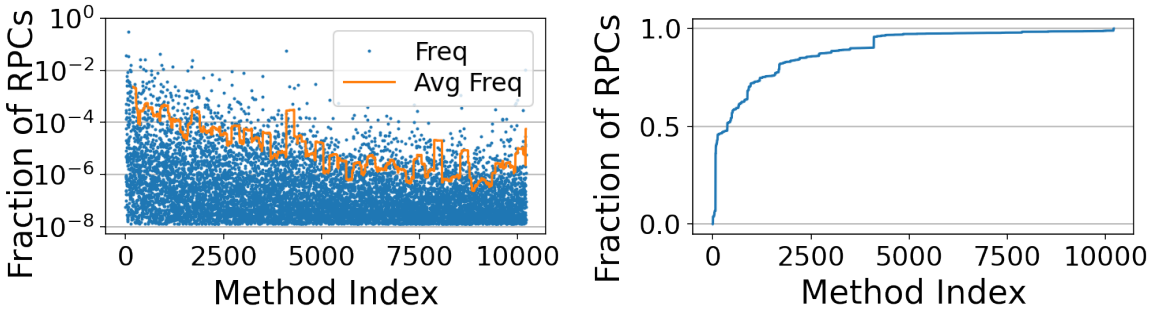
This growth rate puts a tremendous demand on network and compute resources, creating major challenges to sustaining this growth.

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.




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

Figure 2.2: **Per-Method RPC latency, sorted by median latency.**




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

Figure 2.3: **Per-Method RPC frequency, sorted by median latency.**

### 2.3.3 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 2.2a shows, for our 10K methods, the RPC completion time (RCT) 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 2.2b 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 1$^{st}$ percentile latency is 657 $\mu$s 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 [50] and RPC optimization [38–40, 51, 52] 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 2.4.

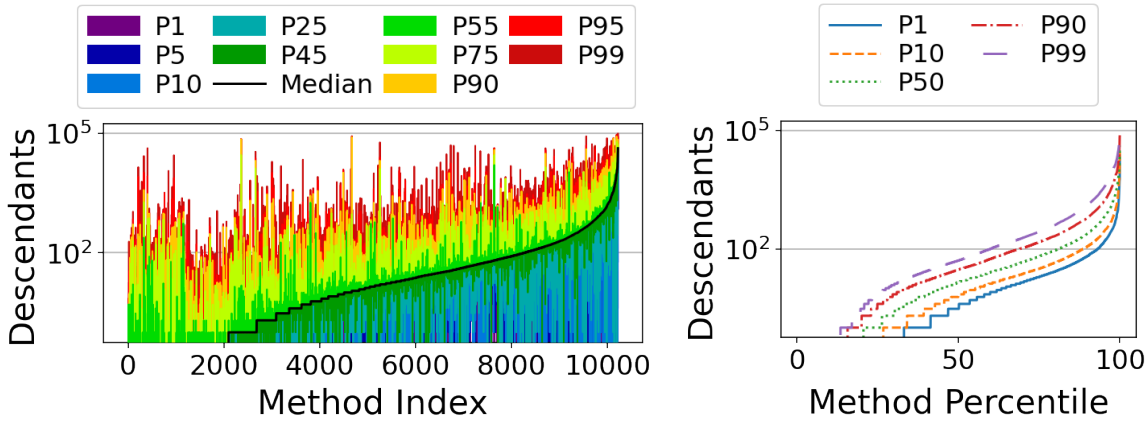
Given the large variation in method latency, we also investigate the popularity of RPC methods. Knowing the popularity distribution tells us the expected benefit of optimizing a small number of methods. Figure 2.3a shows the popularity (relative frequency) of all 10K methods, and Figure 2.3b shows the popularity CDF; both graphs are again sorted in latency order.

From Figure 2.3a 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.

### 2.3.4 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




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

Figure 2.4: **Per-Method Number of Descendants**

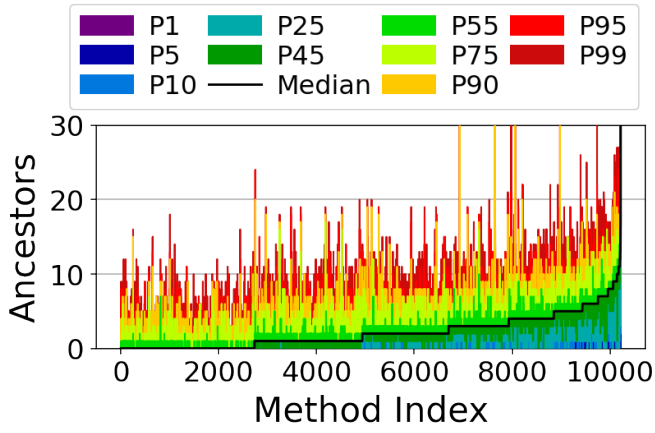
number of descendants shows the scale of distributed computation performed by an RPC, and the number of ancestors provides insights into how the properties of RPCs change as they get deeper into the call graph of a root RPC.

Figure 2.4 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.

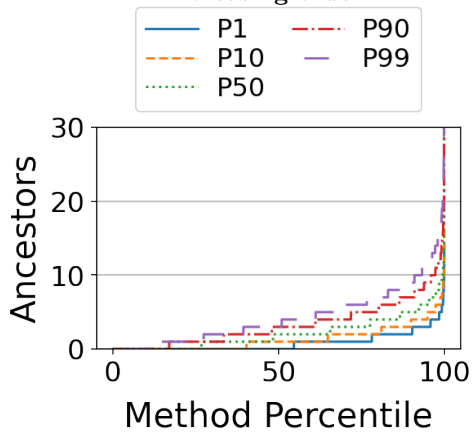
Figure 2.5 shows the number of ancestors for each invoked method, *i.e.*, the return distance from a called method to the root RPC in the tree. Compared to the number of descendants, the number of ancestors for a given invocation is much smaller, implying that the typical RPC call tree is wider than 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.* [17] 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.




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

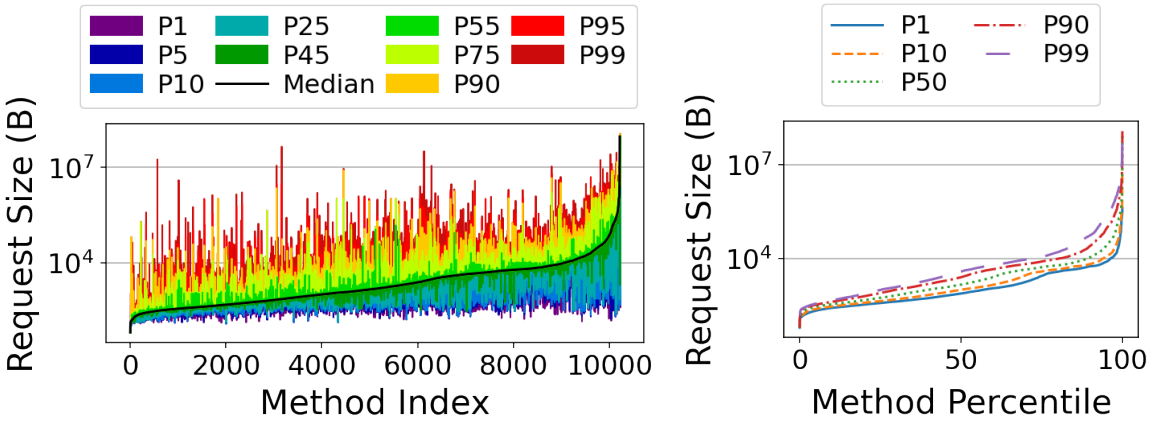



(b) **CDF of ancestor count**

Figure 2.5: **Per-Method RPC Ancestors.**

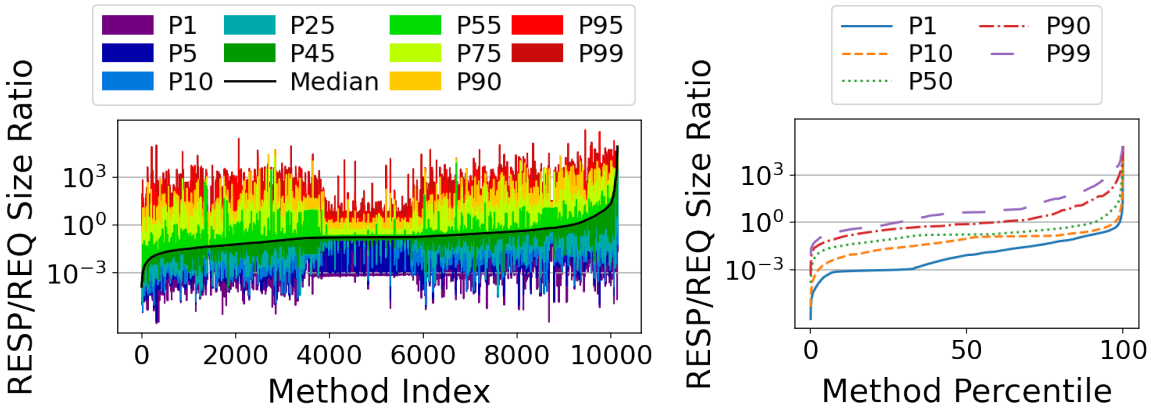
Huye *et al.* [53] report properties about the number of service blocks in the request workflows for a few of Meta's internal applications, and they similarly find that these service graphs are much wider than they are deep. Their P99 depth ranges from 5–6 and their maximum depth ranges from 9–19, and 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.* [50] 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.




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

Figure 2.6: **Per-Method Request Size**




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

Figure 2.7: **Per-Method Response/Request Size Ratio**

### 2.3.5 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.

Figure 2.6 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.

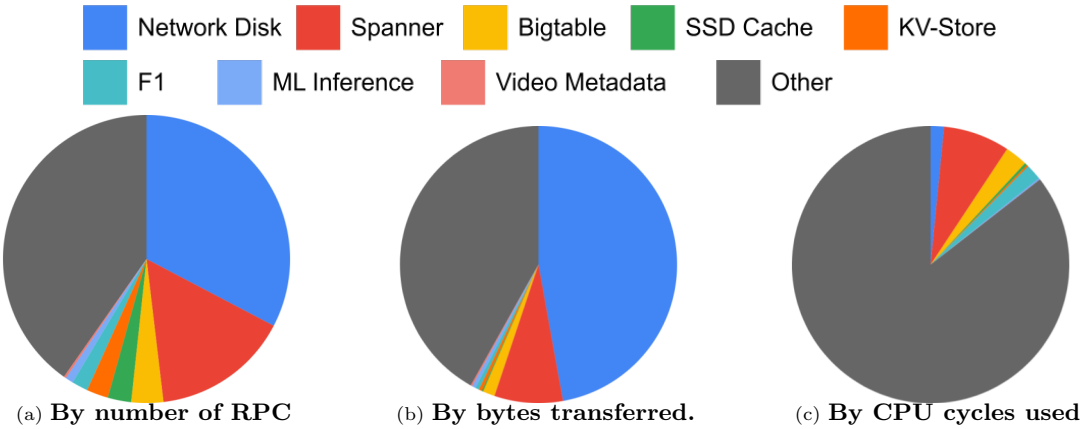
Although most RPCs are small, most methods have a large heavy tail. For example, P90 request

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 [44], 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 2.6 are similar to the distributions reported by prior studies. For example there are two studies of KV-Stores from Meta [54, 55]. 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 2.6, and the Meta distributions for KV-Stores show median values that are on the smaller side of median values shown in Figure 2.6. This is to be expected given that KV-Stores typically persist small amounts of state for otherwise stateless applications [18, 56].

When compared against studies of network flows that identify elephants and mice [7, 9, 57], 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 2.7 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.




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

Figure 2.8: **Fraction of top RPC services.**

### 2.3.6 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 2.8 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 2.8b shows that the distribution of bytes 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.

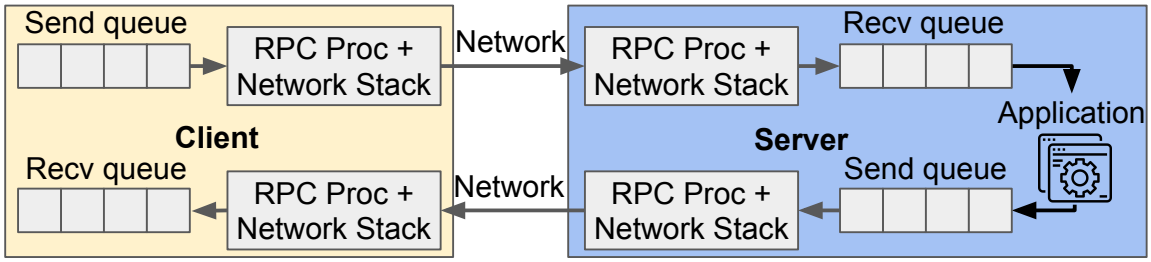



Figure 2.9: **Components in RPC latency breakdown.**

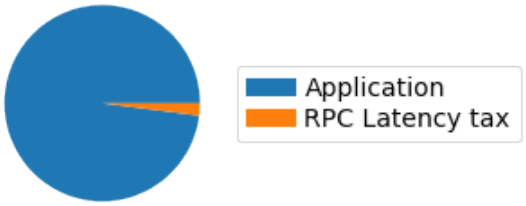
## 2.4 RPC Latency

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.

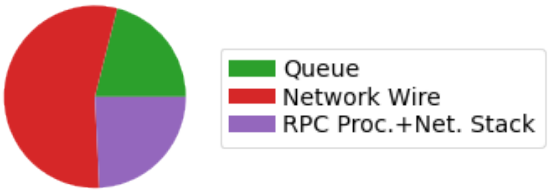
### 2.4.1 RPC Components

We measured the latencies of different RPC components with Dapper. Although different RPC stacks may have different structures, Figure 2.9 illustrates the major components of the RPC stack that we measured, which are described in more detail below.

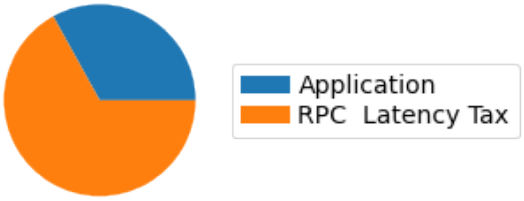
- **Client Send Queue:** Client-side RPC code places requests in a queue where they wait for transmission when local CPU and network resources are available.
- **Request RPC Processing and Network Stack:** This includes the processing and serialization latency for the RPC packets, i.e., the latency taken 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 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.



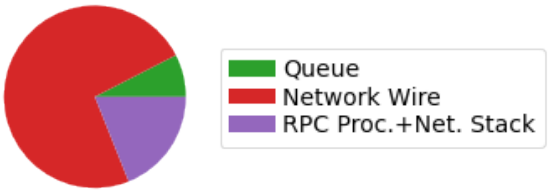

(a) **Fleet-wide Latency spent by RPC Tax.**




(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.**

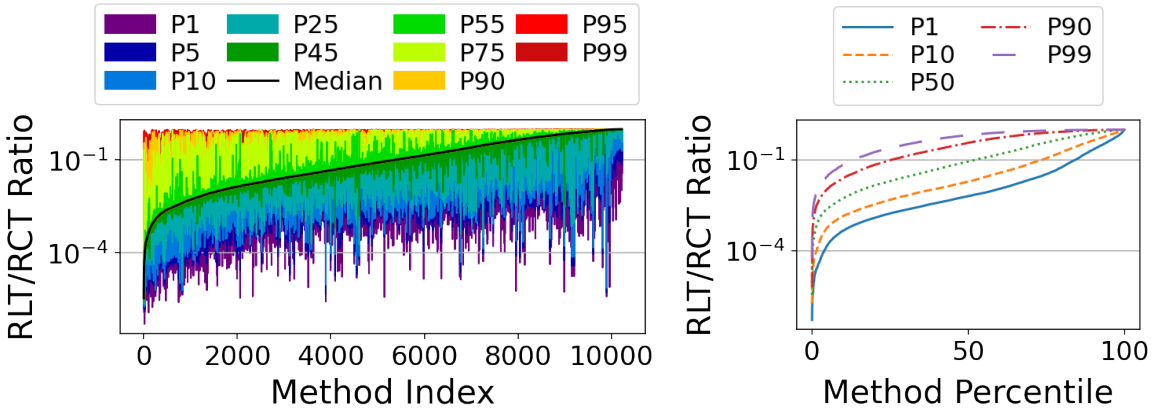
Figure 2.10: **RPC Latency Tax**

- **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).

### 2.4.2 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.

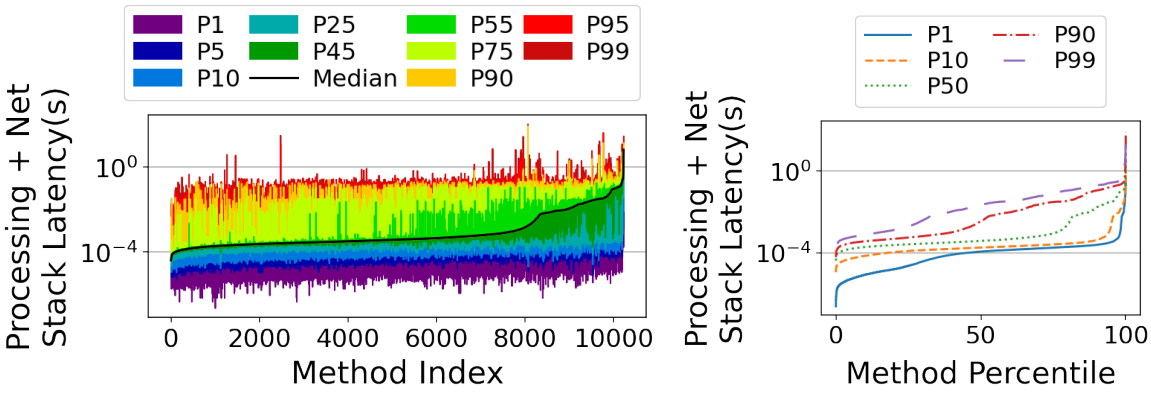



(a) **Distribution Heatmap** (b) **CDF**

Figure 2.11: **Per-Method Ratio of the RPC Latency Tax (RLT) to RPC Completion Time (RCT).**

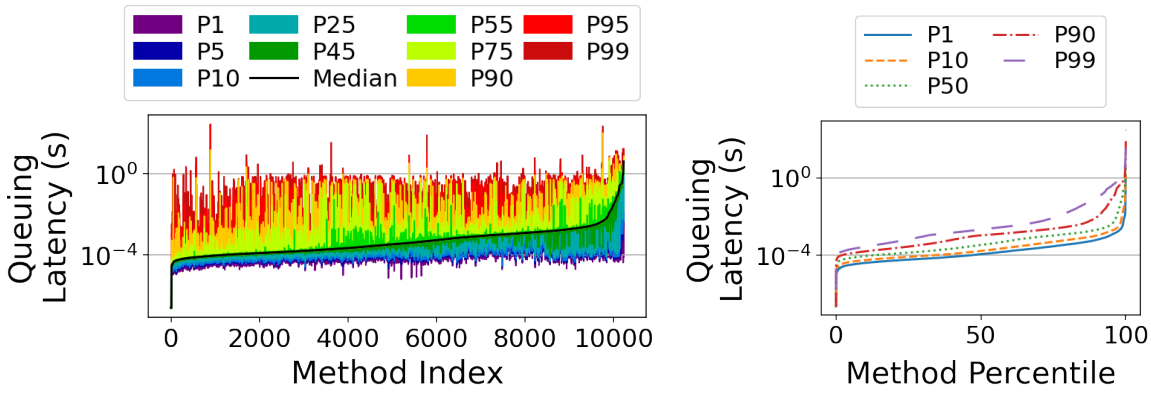
**Application-processing time dominates but RPC tax can be significant.** Figure 2.10a gives an overview of the average RPC latency tax across all RPCs. Overall, the average tax is only 2.0% of the total completion time. Of that 2.0%, Figure 2.10b shows that the network accounts for roughly half (1.1% 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 2.10c and 2.10d. 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 2.11 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 $90^{\text{th}}$-percentile RPC tax is 96%. The $99^{\text{th}}$-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 2.12: **Per-Method RPC Latency Distribution for Network Wire (RW) and RPC Processing and Network Stack (RN).**




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

Figure 2.13: **Per-Method Queuing Latency.**

**RPC tax breakdown.** To quantify the relative contributions of the different RPC tax components, Figure 2.12 shows a per-method breakdown of the latency distribution for combined Network Wire (RW) and RPC Processing and Network Stack (RN). Figure 2.13 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 2.12 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.

| 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 2.1: **RPC services in this study**

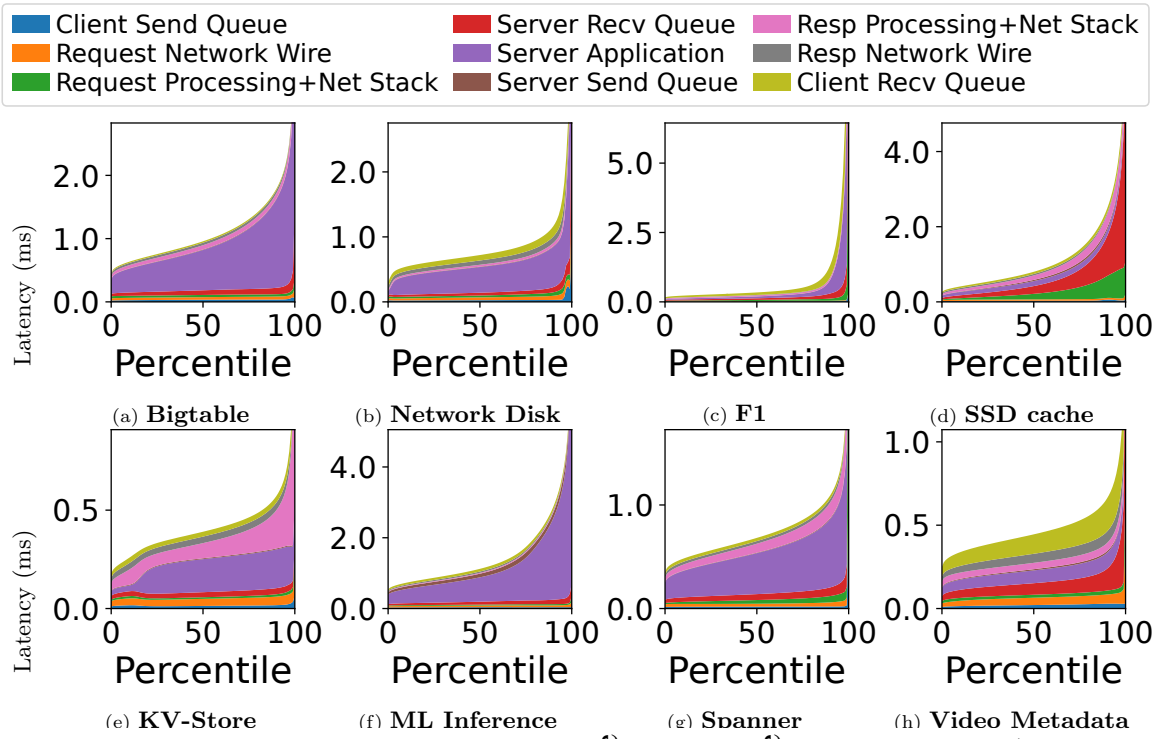
Queuing latency is also a significant contributor to the RPC tax. Figure 2.13 shows that queuing latency is high at the tail, although on the whole it is comparable to the combined RPC processing and network latency. Half of the methods have median and P99 queuing latencies under 360 $\mu$s and 102 ms, respectively, compared to median and P99 latencies under 398 $\mu$s 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.

### 2.4.3 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 2.1. These services can be categorized into three 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.

#### 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 2.1, and (2) from clients located in the same cluster and datacenter as the server. Figure 2.14 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 2.4.1.




| | 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 2.15: **Percent improvement of tail latency (P95) with a what-if analysis.**

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 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.

**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 2.15 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.

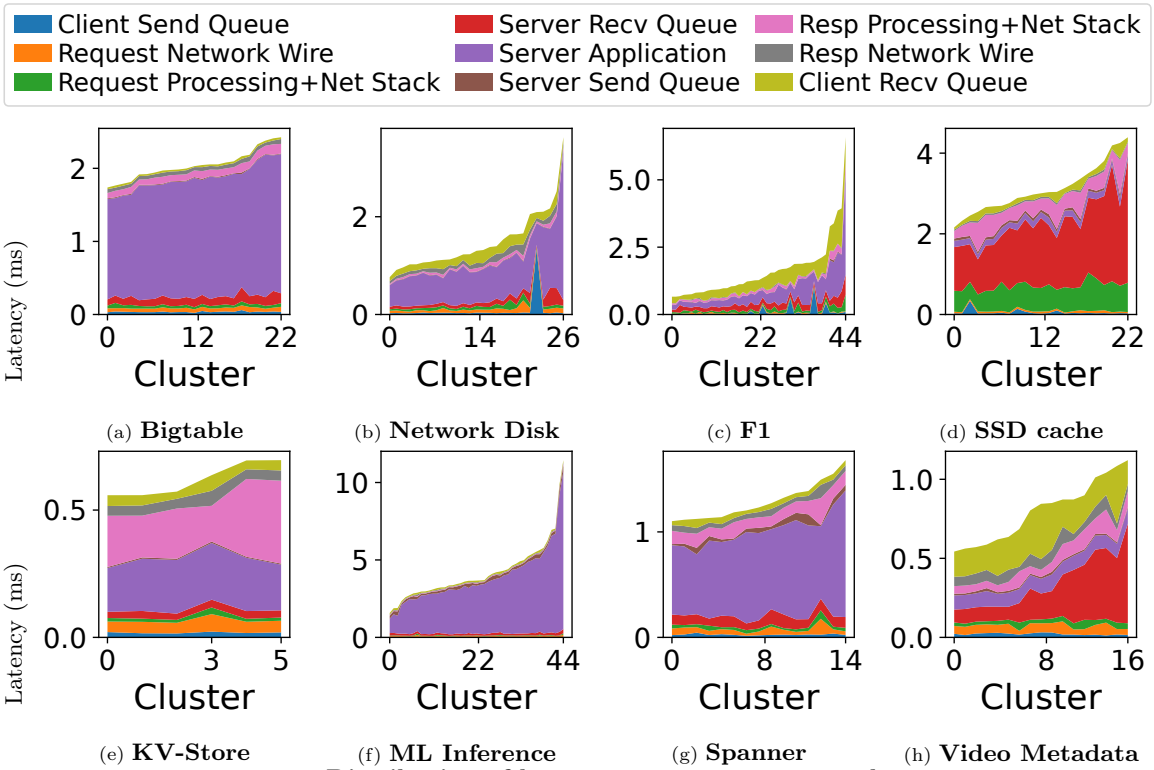
The key findings of this experiment are that, not surprisingly, the major cause of tail RPC latency in the datacenter differs among RPCs and types of applications. Prior work has focused on reducing the latency of the computation of the RPC stack [38] or the network [58] to improve RPC times. In contrast, 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.

**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 2.4.3, 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 2.16 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 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,




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

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

Figure 2.16: **Distribution of latency components across 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*.

### Exogenous Variables Affecting Latency Variation

Figure 2.17 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 2.2). 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 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).

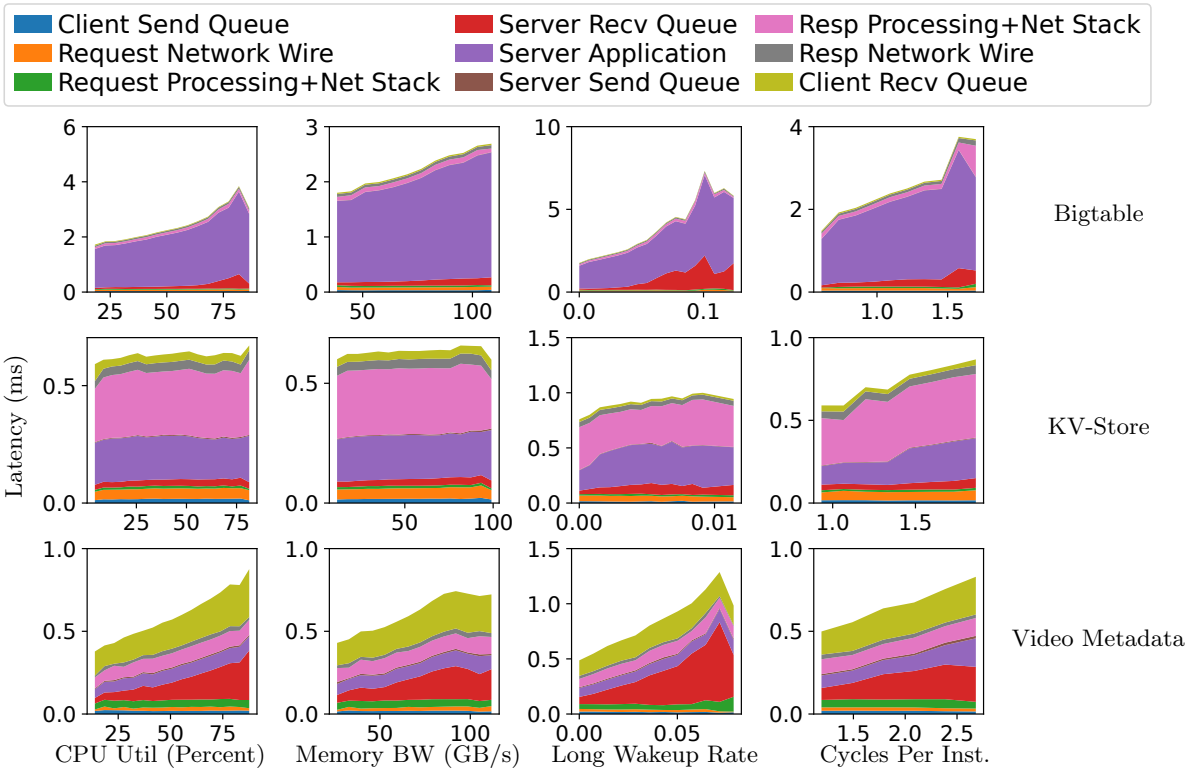



Figure 2.17: **Relation between exogenous variable and latency components.**

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 2.18 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

| 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 2.2: **Exogenous variables**

applications but for different RPCs.

#### 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 2.19 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 2.19 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.

## 2.5 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.

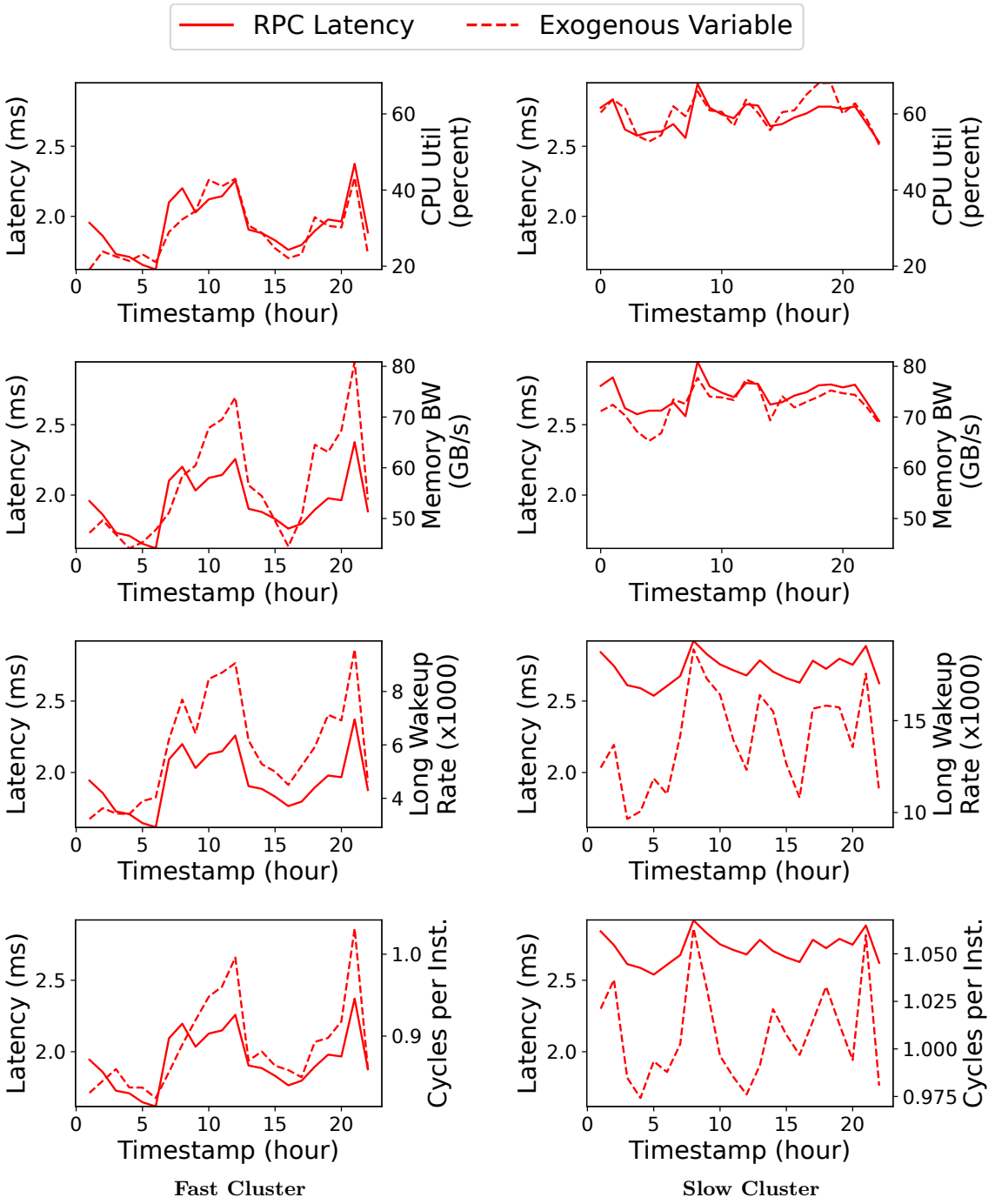



Figure 2.18: **Comparison of exogenous variable and latency between different clusters (Bigtable)**

### 2.5.1 CPU Cycle Breakdown

Figure 2.20 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

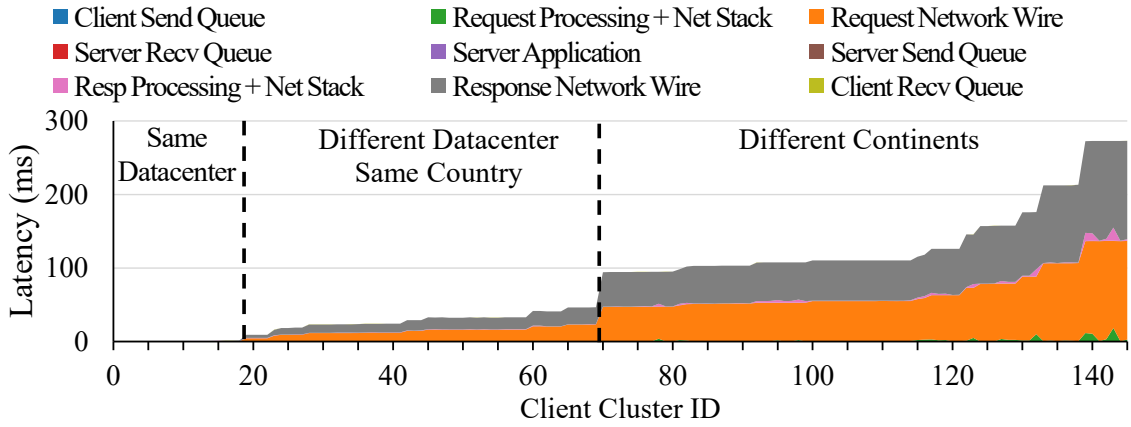
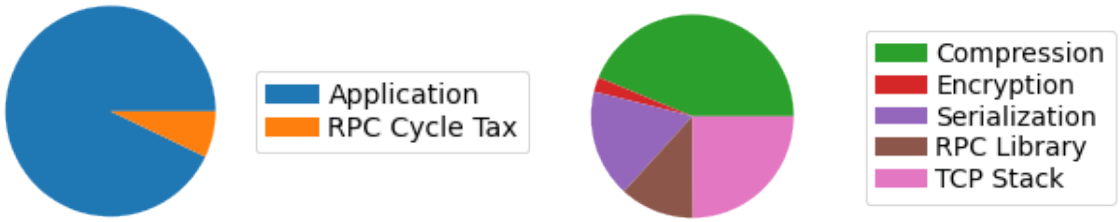
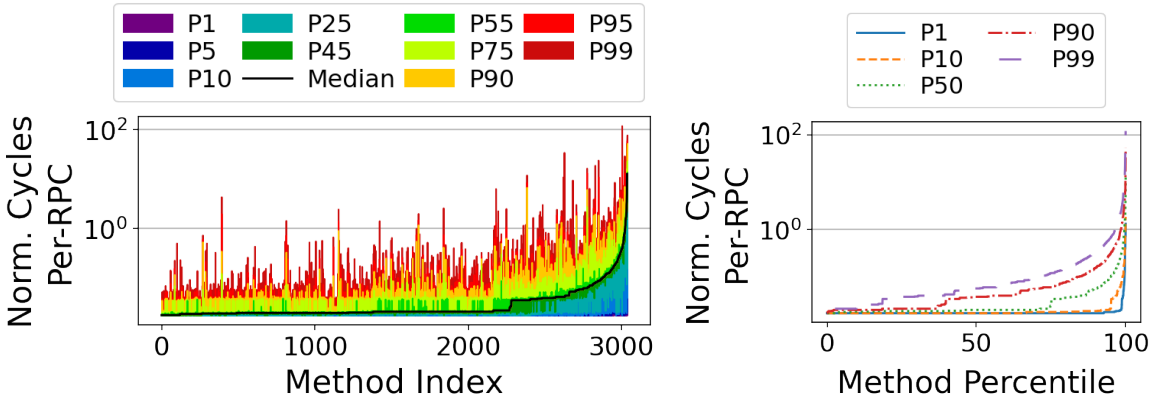



Figure 2.19: **Spanner Cross-cluster latency breakdown.**




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

Figure 2.20: **RPC Cycle Tax.**




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

Figure 2.21: **Per-Method RPC CPU Cycles.**

networking and serialization, at 1.7% and 1.2% of all total cycles, motivating research on serialization offload [44, 45].

### 2.5.2 Fleet-Wide CPU Cycle Variation

Figure 2.21 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 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.

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 [41, 42, 59]. If RPC processing times are not known in advance — which is not always possible because processing-time prediction is a hard problem in general [60] — 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.

### 2.5.3 Load-Balancing Resources

RPC load balancing determines how load is distributed across servers. Figure 2.22 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 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.

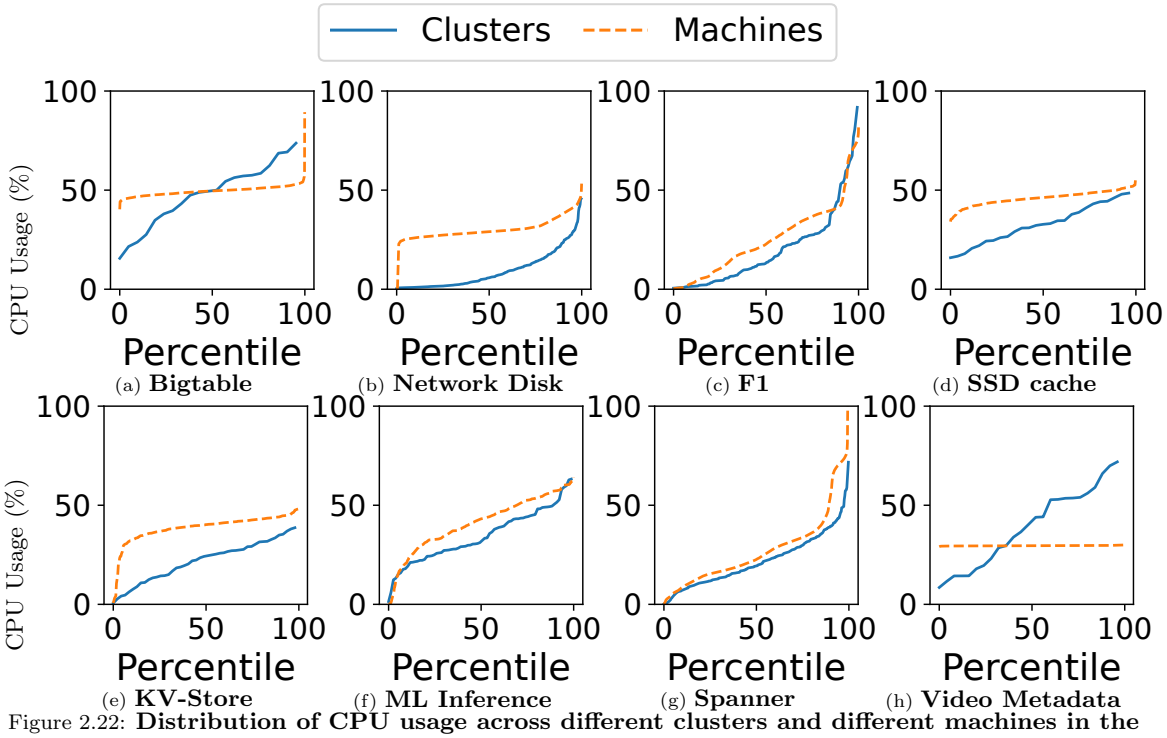



Figure 2.22: **Distribution of CPU usage across different clusters and different machines in the same cluster.**

Conversely, the dashed line in Figure 2.22 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.

### 2.5.4 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 2.23 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 [61] 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,

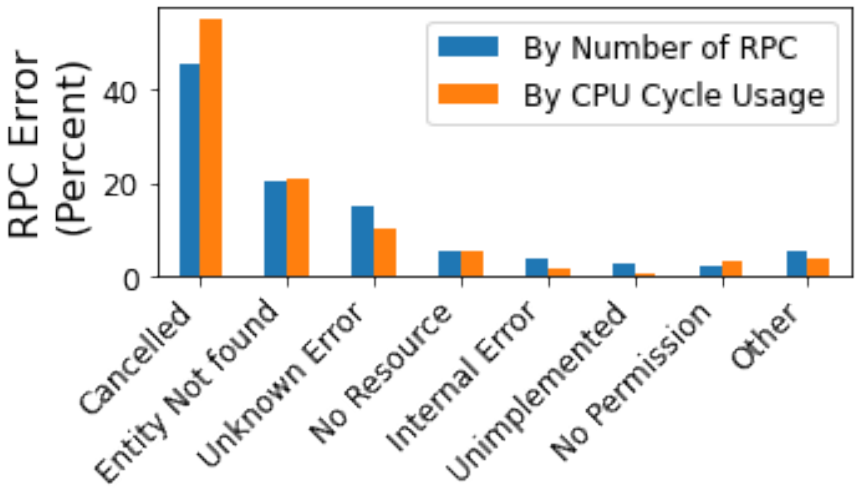



Figure 2.23: **The relative percentage of different RPC error types.**

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.

## 2.6 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.

### 2.6.1 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, 51, 62].

**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 [58, 63, 64], 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 [65–68], 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 $90^{\text{th}}$-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.

### 2.6.2 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 [38, 69], but in reality, future research in service-specific optimizations targeting each of their main bottlenecks will be crucial as well.

### 2.6.3 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, 51, 62] 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 2.20a 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 2.8) — Network Disk and Spanner — which are data intensive. This demonstrates the potential for accelerating data movement [70, 71].

**Method-specific hardware optimizations.** Accelerators must cover a significant fraction of the CPU cycles consumed by the fleet to provide cost-efficiency benefits [70–72]. 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.

### 2.6.4 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 [22–24, 38, 73, 74]. However, we have focused on RPC over TCP and leave the study of RDMA to future work.

# Chapter 3

# Accelerating data durability of remote storage systems

## 3.1 Introduction

The benefits of economy of scale and the emergence of cloud computing have caused an increasing number of enterprises to move their workloads to hyper-scale cloud data centers, with an estimated annual growth of about 14.6% during the period 2017–2022 [75]. Today, most of the computation takes place in these hyper-scale cloud data centers—performing more than 89% of the computation world-wide in 2018 [76].

These data centers host workloads ranging from time-critical, interactive jobs (e.g., online data-intensive (OLDI) workloads [14], RAMCloud [77, 78], and financial analysis [79]) to long-running, batch jobs (e.g., MapReduce [80] and machine-learning training [81]) with large memory footprints. In most of these workloads, data is typically managed and maintained in a persistent way across multiple servers, with clients accessing and updating this data remotely over a network of interconnected switches, using remote procedure calls (RPCs). During each invocation of an RPC, the request is processed by the client's IO stack, the network of intermediate switches, the server's IO stack as well as the request handler on the server. Thus, the latency of an RPC is significantly affected by the processing time of each of these stages. As the computation performed by modern workloads is

dominated by these RPCs, i.e., read and update requests, the time it takes to access remote data is of major consideration when deploying workloads on modern data centers [82–86].

These RPCs can be either *synchronous* or *asynchronous*. Though, asynchronous RPCs can enable clients to continue execution while updates are being processed at the remote server; yet, building such applications is quite challenging, especially "at scale, when a typical end-to-end application can span multiple small closely interacting systems" [83]. In contrast, applications using synchronous RPCs are easy to write, tune, and debug—Google is known to strongly prefer a synchronous programming model [87, 88]. Therefore, in this paper, our aim is to improve the performance (specifically the tail latency) for synchronous RPCs by minimizing the access time to remote persistent data.

Recently, as programmable network devices become available [89], a trend is to offload application logic to those devices. This way, a large fraction of the procedure, including server's network stack and processing time, is no longer handled by the server but accelerated by those network devices. This newer computation scheme is known as in-network compute, spanning a wide range of applications, such as query processing [90, 91], key-value stores [11, 92–94], data aggregation [95–97], and even computational-intensive machine-learning tasks [98–100].

Though promising, in-network computing mainly mitigates the latency of computational tasks and requests that do not change the server state (such as read queries); the data persistence is still maintained by the servers, and update requests still need to traverse the entire network and server's IO stack to complete the update. Therefore, as in the original case, the client needs to wait for an entire round-trip time (RTT)—for an acknowledgement from the server—before it can proceed to the next step.

To minimize the request processing time on the servers, data centers [101, 102] are deploying new persistent memory (PM) technologies, such as Intel's Optane [3] and NVDIMM [103]. Compared to traditional storage devices (e.g., SSD and HDD), PM provides high-speed and direct, byte-addressable access to persistent data, while bypassing OS indirections (e.g., file systems). PM reduces the server's storage latency by 10∼50× [104–106], thereby enabling software systems (such as databases [107–110] and key-value stores [111–113]) to perform at much faster speeds. Even though the integration of PM significantly reduces the request processing time at individual servers, the network is still a dominant factor when processing these requests in a data center—causing clients to stall for a complete RTT. Moreover, as the network is a shared resource, the contention for bandwidth, switch queues, and links can lead to variable delays and long tail latencies [61, 62, 114–120]. We identify that the fundamental

cause of the limitations of in-network computing is that operations are stateless, being unable to accelerate a stateful, persistent operation on the server. On the other hand, persistent memory in the server improves the performance of persistent updates but still puts the network and server network-stack latency on the critical path.

We found that it is possible to expose the persistent state to the network and persist update requests in-network. Therefore, we introduce the notion of *in-network data persistence*, which enables a sub-RTT latency when processing update requests. To expose data-persistence domain to the network, we *log updates* in the network using persistent memory and send acknowledgements to clients as soon as a request enters the persistent domain. The update requests are then forwarded to the server, but this way, the server processing happens off the critical path. As the requests have entered a persistent state before being processed by the server, the client can now proceed before the server acknowledges.

In this work, we design and implement PMNet, a mechanism necessary to provide *persistent logging support* in programmable network devices. However, designing PMNet has many challenges. First, how can a network device track requests and persistently maintain their state? Second, given the requests have been persisted in the network, how can the system recover after a failure? Third, how can PMNet maintain the same application-level ordering guarantees with in-network persistence? Next, we describe our key insights.

**Persistent logging** PMNet uses a simple protocol to ensure that updates are logged persistently in the network device with sub-RTT latency. First, PMNet mirrors the incoming update requests while they are traversing the network device. It logs the update requests in its PM. Second, as the requests have already entered a persistent domain of the network device, PMNet immediately sends an acknowledgement to clients, allowing them to move forward. Therefore, compared to the original scenario, the latency is significantly reduced as the client no longer needs to wait for the whole RTT. Third, PMNet invalidates the logged entry upon reception of an acknowledgement from the server, which indicates that the server has completed the request.

**System recovery** In case a failure happens in the persistence domain (i.e., the network device and/or server), PMNet needs to ensure that logged entries are reflected on the server. When the system is up again, PMNet resends the logged requests so that servers can redo them in the *same*

*order* as they were sent. As such, the server can recover to a consistent state with the logged requests.

**In-order delivery** PMNet always maintains the ordering of the original system. As the logged updates are reflected later on the server, one may think that a client will read a stale value from the server. However, we observe that oftentimes large-scale workloads optimize for independent clients. For example, in a Twitter workload using Redis as the backend [1], the clients update tweets and followers without maintaining any order. Still, PMNet provides ordering guarantees when there is a strict ordering requirement within multiple clients. These workloads enforce application-level ordering using synchronization primitives to ensure that only one client can update a critical value. For example, a TPCC workload [2] puts the modification of the stock price in a critical section using locking primitives. PMNet treats the lock operations in a critical section as regular read requests and forwards them to the server. Therefore, the ordering is enforced on the server and subsequent lock requests from other clients fail on the server, but subsequent update requests from the same client operate with sub-RTT latency by persisting these update requests in the network device.

We implemented PMNet in an FPGA-based programmable switch (PMNet-Switch) and a NIC (PMNet-NIC). Our evaluation shows that both designs provide a significant benefit over the baseline system. However, the latency difference between PMNet-Switch and PMNet-NIC is negligible as the round-trip time is dominated by the server network stack and the server processing time. On top of PMNet, we further integrate additional functionalities. *(1) PMNet-Switch with caching:* We demonstrate that our logging mechanism for update requests works coherently with a prior work that proposed to cache read requests in a switch [11]. *(2) PMNet-Switch with replication:* We develop an in-switch replication mechanism that builds upon PMNet's logging protocol. In summary, we make the following contributions:

- We expose data persistence to the network to improve the performance of update requests. We implement PMNet using a programmable data-plane device, integrated with a persistent memory that logs incomplete update requests.

- We adapt common PM workloads, including Intel's PMDK-based key-value stores [121], a PM-optimized Redis database [110], Twitter [1], and TPCC [2] to PMNet. Our evaluation shows that PMNet improves the throughput of update requests by 4.31× and the 99th-percentile tail latency by 3.23× in these workloads.

- We also demonstrate that PMNet improves read-caching and state-replication latency by 3.36× and 5.88× respectively, over traditional, baseline systems.

## 3.2 Background and Motivation

In this section, we first discuss the performance bottlenecks in performing update requests in persistent applications and then describe our proposed solution.

### 3.2.1 Synchronous Programming Model

Data centers host a wide range of workloads, such as online data-intensive (OLDI) workloads, RAMCloud, and financial analytics [14, 77–81], where data is typically managed in a persistent way through multiple servers, and accessed via queries. If we categorize these requests, there are mainly two types: synchronous and asynchronous. A synchronous request guarantees its completion at the server by blocking the client until the server responds. In comparison, an asynchronous request lets the client proceed immediately while the request is being delivered and processed at the server. However, the client risks losing data in the face of common failures, such as network packet loss. When such failure happens, the application needs to ensure that the clients and the servers remain in-sync—complicating the design and development of the application. Therefore, programmers usually prefer the synchronous programming model [83, 87, 88, 122].

### 3.2.2 Mitigation of the Synchronous Overhead

Although the synchronous model alleviates the programming burdens, it places the entire query RTT on the critical path of the application, as future requests are blocked until the in-flight request has been processed by the server. Figure 3.1 demonstrates the steps involved in query processing, including the client's network stack, the network latency, the server's network stack, and the server's request processing. The dashed arrows demonstrate the RTT of processing a single request. Further breakdown in Figure 3.2 shows that server-side latency, including the server's network stack (in the kernel) and request processing time (in the user-space), makes up the majority of the overhead (70% on average). To mitigate the synchronous overhead, one of the promising solutions in the literature is to offload tasks to the network. By placing computational devices—such as programmable switches [123] and SmartNICs [124]—on the network path, a large portion of the server's processing and network stack latency can be eliminated. Examples of in-network computing include load balancing [125, 126],

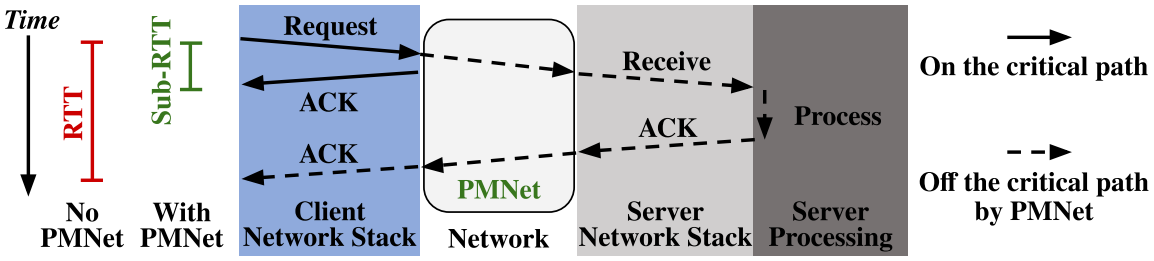



Figure 3.1: Round-trip time (RTT) of a single request.

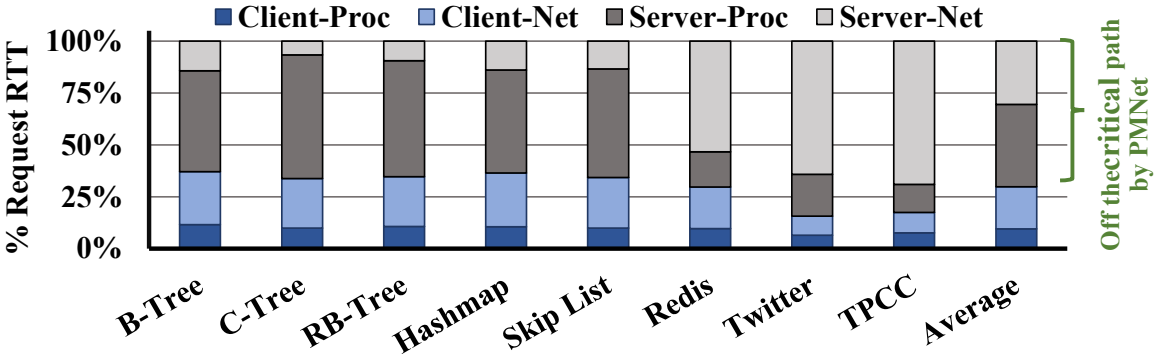



Figure 3.2: Latency breakdown of an update request.

congestion control [127, 128] and packet scheduling [129, 130], query processing [90, 91], key-value stores [11, 92–94], and machine-learning acceleration [98–100].

However, in-network compute only handles stateless requests that do not change the persistent state. In case of an update request, the communication and processing overhead remains. Fortunately, recent advancement in memory technology provides an alternative, high-performance storage system, persistent memory (PM) [3]. By managing persistent data on PM, the server can perform update requests more efficiently and reduce the server-processing latency. For example, databases and key-value stores [107–113], and PM-optimized file systems [131–138] optimize their data management to take advantage of PM's high performance. Despite these improvements, the network stack and the processing time remain on the critical path. Even with an optimized network stack, such as using libVMA [139] that enables applications to bypass the server's network stack, the server-processing time is still a major overhead (we evaluate an optimized network stack in Section 3.6.2).

In this work, we ask the following question: *how can we move the server's network stack and processing time off the critical path and process the client requests in sub-RTT?* A promising solution is to log and persist the update requests ahead of time before they enter the server stack and processing time. A dedicated logging module can bypass the majority of system overhead. Unfortunately, a dedicated software module can introduce additional CPU and memory utilization (evaluated in Section 3.6.2), and a dedicated hardware module needs a redesign of the CPU architecture, making

fast deployment challenging. The trend in data centers is to move dedicated logic into NICs and switches [11, 90–100]. This method is effective in handling network traffic, without putting extra load on the server. Following this trend, our *goal* is to *design and prototype* an efficient logging mechanism using programmable network devices.

### 3.2.3 In-Network Data Persistence

For data to become persistent in the network, we need to extend the data-persistence domain from within servers to the network. By maintaining the persistent state of on-going requests in the network, an update request can become equivalently persistent *before* having been processed by the server. In case of a failure on the server, the already persisted requests can be resent from network devices and re-applied to the server for recovery. As the solid arrows in Figure 3.1 demonstrate, clients no longer need to wait for the entire RTT for servers to process the requests and reach a persistent state. Instead, they can proceed once the requests have entered the persistence domain of the network. Consequently, the entire server-side latency (as pointed out in Figure 3.2), including the request handler's processing time and the network stack latency, can be placed off the critical path. Therefore, persisting these requests in the network can significantly improve performance by taking the server off the critical path.

## 3.3 High-level Ideas

We present PMNet, a PM-integrated programmable network device that realizes in-network data persistence. By keeping a persistent copy of in-flight requests on the network device, the update will end up in a persistent state *before* having been received and processed by the server. In case a server fails (e.g., due to a power outage or kernel crash), the persistent copy of the request can act as a redo log for the in-flight requests when the server recovers from failure. At a high level, the design and implementation of PMNet have three major challenges: (1) How can PMNet move the network and server processing time off the critical path by logging the requests? (2) How can the system recover using the logged requests? (3) How can the existing client and server applications maintain ordering after having PMNet integrated? Next, we describe the high-level ideas of PMNet that address these challenges.

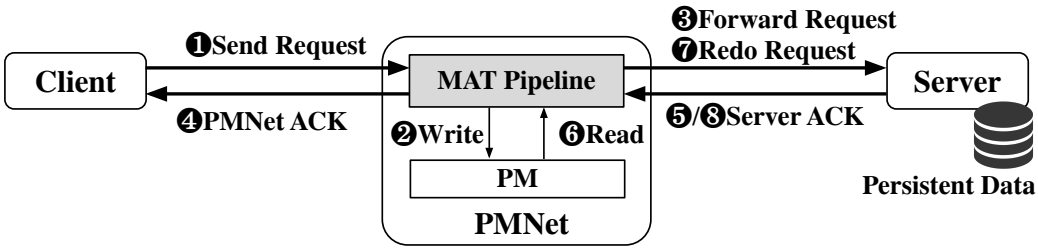



Figure 3.3: Request logging and system recovery in PMNet.

### 3.3.1 Persistent Logging

With the integration of PM, PMNet can maintain a copy of in-the-flight update requests in PM, to serve as a redo log for the server. This way, the request reaches a persistent state before it has been processed by the server (Figure 3.1). PMNet then sends the acknowledgement of the request to the client once it has been persisted in the network device's PM, rather than waiting for the server's acknowledgement. Figure 3.3 shows the workflow of the request logging procedure: Upon receiving an update request, PMNet writes it to the integrated PM (step ❶-❷). While the request is being written to PM, PMNet forwards it to the destination server (step ❸). After the request becomes persistent, PMNet sends an acknowledgement (`PMNet-ACK`) to the client, indicating that this request has entered a persistent state (step ❹). When the server has actually processed the request, it acknowledges (`server-ACK`) PMNet to invalidate its copy and reclaim the log entry for future use (step ❺). On the client-side, upon receiving an acknowledgement from PMNet, the client proceeds without waiting for the completion of the request at the server (details in Section 3.4.2). PMNet can work as a *switch* as well as a *NIC*, which we refer to as *PMNet-Switch* and *PMNet-NIC*, respectively.

### 3.3.2 System Recovery

The next challenge is to recover the server after a failure. Figure 3.3 shows the recovery procedure. Upon detection of a server failure (e.g., through heartbeat signals), PMNet reads the logged requests from the device's PM (step ❻) and resends them to the server (step ❼). PMNet ensures that the server commits the resend requests in the original order by adding an extra *sequence number* in the header (details in Section 3.4.5). Once the server has committed a request, it notifies PMNet to invalidate the log entry (step ❽). Besides this sequence number, the update requests need additional information to ensure correct in-network logging and recovery. PMNet encodes this information as a new PMNet header (details in Section 3.4.1) to existing network protocols (e.g., IP or VXLAN), and provides software support for the client and server to process this encoding with minimum changes to the source code.

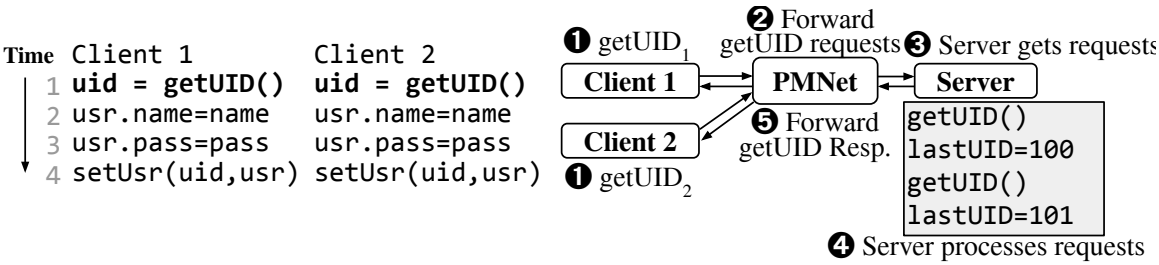



Figure 3.4: No ordering in a Twitter workload [1].

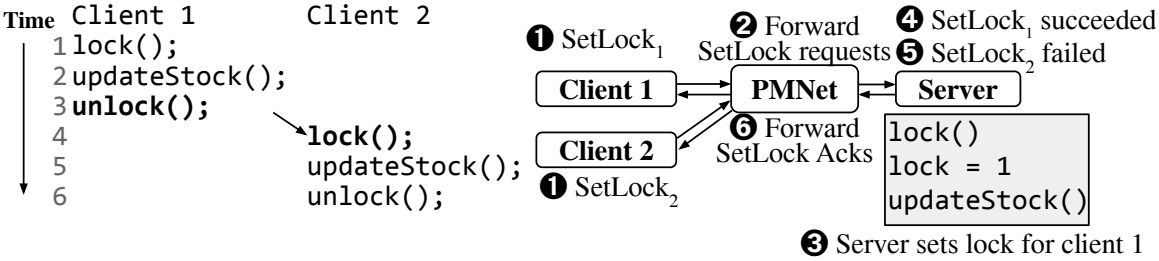



Figure 3.5: Application-level ordering in a TPCC workload [2].

### 3.3.3 In-order Delivery

PMNet guarantees the same ordering as the traditional, baseline systems. We already discussed that PMNet maintains ordering within a client by keeping additional sequence information in the header of the packets from each client. However, one may wonder how PMNet ensures the ordering between multiple clients. Can another client read the up-to-date state from the server, while the update is being logged in PMNet and not yet committed to the server?

PMNet targets the common cases, where the majority of client-server connections are independent. This is unsurprising because the RTT between a client and server is as high as tens of µs—having dependencies (and synchronization) with other clients can inflate the end-to-end latency of requests even further. Therefore, large-scale workloads typically mitigate dependencies and optimize for independent clients. For example, different tasks in a microservice use separate storage backends without dependencies with others [114, 115], worker nodes in distributed machine-learning systems send new weights to a parameter server without synchronizing with other workers [81, 140], and client-independent databases, such as Memcached and Redis, are commonly used as service backends [141, 142]. Figure 3.4 demonstrates two update requests to a shared variable `lastUID` in the *Twitter* workload [1]. There is no ordering constraint among clients, and each client independently executes the `getUID` function (line 1) and uses that `UID` for consecutive requests.

However, in rare cases where ordering must be enforced among multiple clients—i.e., one client cannot update the server until another completes—the client needs to make sure the request has actually

been processed by the server before making any forward progress. These workloads typically use synchronization primitives to ensure the ordering at the application level. The client uses an update request to access the synchronization primitive on the server and acquire the lock on the critical section; other clients are blocked from entering the critical section, thus enforcing an ordering among clients. Figure 3.5 shows this synchronization scheme in the TPCC workload [2], where PMNet directly forwards the locking requests to the server and only *client 1* can access the critical section. And, subsequent update requests from *client 1* can still benefit from PMNet. As lock requests are forwarded to the server, the ordering of lock-acquire/release is enforced. Because lock requests are infrequent, the majority of requests can be logged by PMNet. In our experiment, most workloads are lock-free. Only 13.7% of the requests in the TPCC workload access the locking primitive (i.e., bypass PMNet).

## 3.4 PMNet Design

So far, we have introduced the high-level ideas of PMNet, that extend the persistence domain from servers to the network by logging in-flight update queries in the network devices' PMs, and redo them in the event of failures. This way, PMNet effectively moves server-stack overhead off the critical path of request processing.

PMNet realizes in-network data persistence by augmenting programmable network devices (e.g., switches and NICs) with PM, as the architecture overview in Figure 3.6 shows. To have PMNet fully functioning end-to-end, from the client-server application to the hardware implementation, we introduce four major aspects in this section. First, we introduce PMNet protocol for update requests that can benefit from in-network data persistence, which encodes metadata that is necessary for logging and recovery into a PMNet header (Section 3.4.1). Second, we describe the request processing procedure of PMNet, according to the PMNet protocol (Section 3.4.2). Third, based on the design of PMNet, we further introduce two use-cases of PMNet. One case that integrates PMNet into a replication system (Section 3.4.3), and another case that implements a read cache on top of PMNet's logs (Section 3.4.4). Finally, having with all the details described, we illustrate the recoverability of a PMNet-based system (Section 3.4.5).

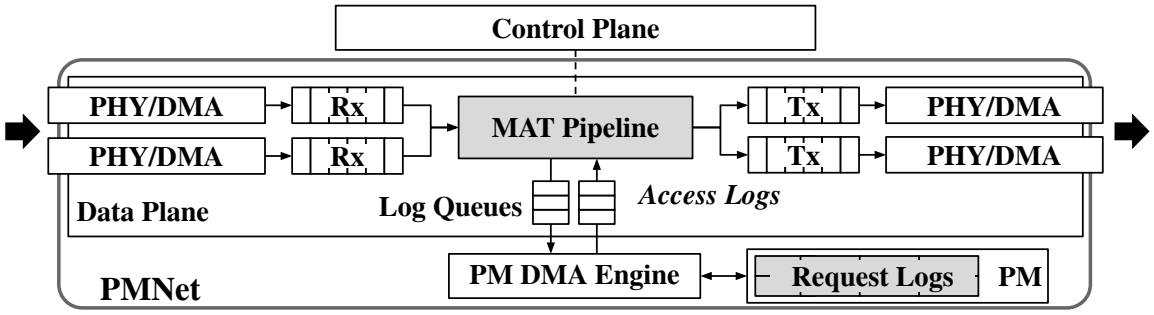



Figure 3.6: A high-level view of the PMNet architecture.

### 3.4.1 PMNet Protocol

We now describe the PMNet protocol, including its packet header, delivery method, and ordering guarantees for queries.

#### Header format

The PMNet header is placed in the application layer of each network packet (i.e., L4) (Figure 3.8). A PMNet header consists of the following fields:

- **Type** (8 bits) differentiates the type of PMNet's requests (details in 3.4.2).
- **SessionID** (16 bits) keeps track of sessions a client sends requests from and differentiates connections among clients.
- **SeqNum** (32 bits) tracks the order of packets sent over a given session, such that the server can process the queries in the original order. Furthermore, the latest `SeqNum` informs PMNet to avoid redoing already completed queries during the recovery (Section 3.4.5).
- **HashVal** (32 bits) is a CRC-32 hash of the entire header that is computed by the sender's network stack. The PMNet uses this hash value to index a packet in the PM of PMNet.

#### Delivery method

The PMNet protocol is built upon UDP, similar to other in-network compute works [93, 98, 143, 144]. To differentiate from other network traffic, PMNet reserves specific UDP ports (range: 51000–52000) and encodes PMNet header into the UDP packet. In case an application originally uses TCP, PMNet's software library converts TCP packets to UDP packets while maintaining a reliable delivery guarantee of TCP (similar to [128]). We present how PMNet protocol ensures the packet ordering and integrity guarantees in Section 3.4.1.

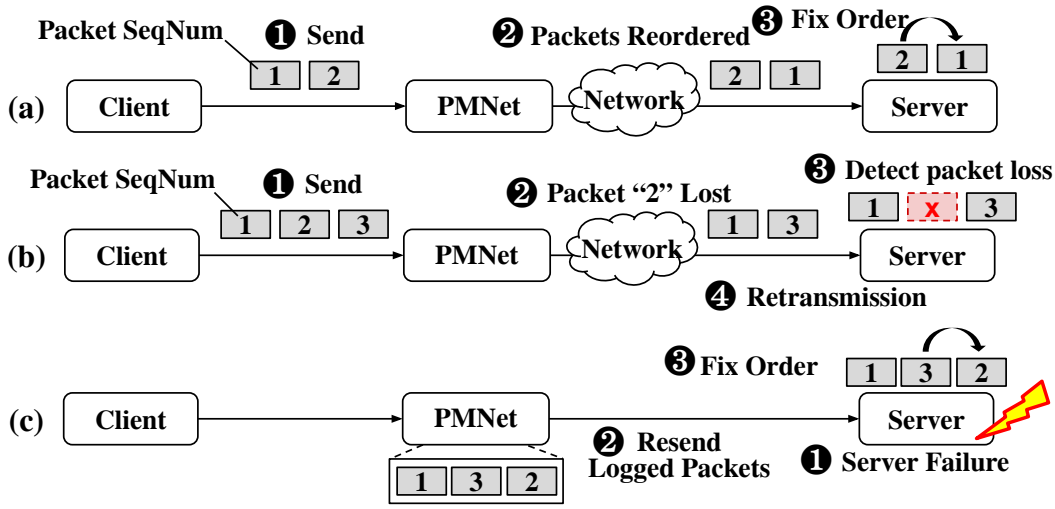



Figure 3.7: Per-client packet ordering guarantee in three cases: (a) reordered packets, (b) packet loss, and (c) failure.

### MTU-sized packets

PMNet obtains the packet size from the UDP header. Though a UDP packet typically has a maximum transmission unit (MTU) of 1.5 kB, a query can be larger than this limit. PMNet's software library transparently divides the queries larger than MTU into smaller packets and uses a sequence number (`SeqNum`) to maintain packet order. PMNet handles MTU-sized packets by sending a per-packet `PMNet-ACK` to the client once each packet has been logged in PMNet. And, the client needs to collect *all* `PMNet-ACK`'s to make sure the corresponding update request has been completely logged in PMNet. The PMNet's software library also tracks the number of `PMNet-ACK`'s in a similar way.

### Ordering guarantees

As UDP does not guarantee packet ordering, PMNet protocol implements such ordering for servers to execute queries from the same client in the *original order*. We discuss the ordering guarantee in three scenarios. (1) During normal execution: Figure 3.7a demonstrates a scenario where the client first sends packets in the original order (step ❶), and during network transmission, some packets are reordered (step ❷). On detection of out-of-order packets, the server's PMNet library corrects the order based on `SeqNum` of each packet (step ❸). (2) On detection of a packet loss: Figure 3.7b demonstrates a scenario where the client sends a series of packets (step ❶), but some packets are lost (packet-#2 in this example) during network transmission (step ❷). The server's PMNet library detects nonconsecutive `SeqNum` (step ❸) and requests for retransmission (step ❹). Section 3.4.2 describes more details about retransmission. (3) During failure recovery: Figure 3.7c demonstrates

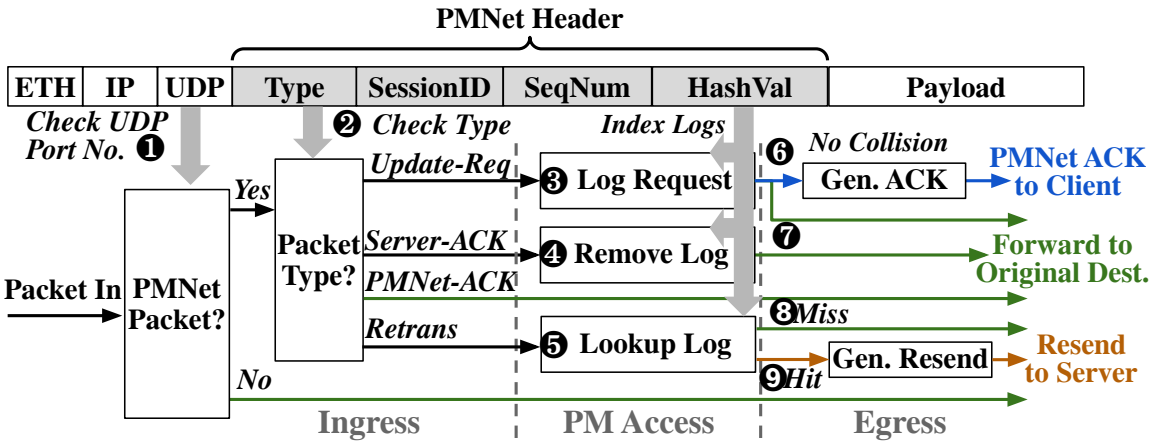



Figure 3.8: The data-plane MAT pipeline in PMNet.

a scenario where the server fails (step ❶) and PMNet retransmits the logged packets to the server for recovery (step ❷). Similar to scenario (1), the server reorders the packets on reception (step ❸). In conclusion, although the network may reorder or lose packets, PMNet protocol guarantees their ordering and delivery.[1]

**Multi-client ordering** PMNet enforces multi-client ordering at the application level by forwarding synchronization requests directly to the server (described in Section 3.3.3). Doing so allows the server to enforce ordering across critical sections of the application code, while the code within the critical sections can still benefit from PMNet's logging.

### 3.4.2 PMNet Request Processing and Log Management

#### Packet handling

In PMNet, a match-action table (MAT) pipeline (Figure 3.6) handles the following types of PMNet packets, distinguished via the "`Type`" field, along with other non-PMNet packets.

- **Update requests from the clients (`update-req`).** PMNet needs to maintain the persistent state of these queries in order to recover. Therefore, upon reception of a packet that belongs to an update request, PMNet immediately forwards the packet to the destination server, and in the meantime, logs the packet in the persistent memory. The `HashVal` in the PMNet header serves as the index to the log entry. Once the whole packet has been persisted to the network device's PM, PMNet sends a `PMNet-ACK` back to the client, as the packet has reached a persistent state. Note that in cases where the device PM is full or the `HashVal` of the packet collides with an

[1] Note that handling reordering at the network level is rare as datacenter networks typically use flow-consistent load balancing (e.g., ECMP [38]).

existing entry, PMNet directly forwards the packet to the destination server without logging it (or acknowledging the client).

- **Bypass request from the client (`bypass-req`).** For purposes such as read request and synchronization, where the client does not need to receive an early acknowledgement from PMNet, the client sets the packet type to `bypass-req`. PMNet directly forwards it to the destination without logging. As a result, these packets do not enter a persistent state until processed by the server.

- **ACK from another PMNet (`PMNet-ACK`).** In a system with multiple PMNets, a PMNet may receive a PMNet-ACK from another PMNet. In this case, PMNet directly forwards the packet along its path to the destination.

- **ACK from a server (`server-ACK`).** Upon reception of a `server-ACK`, PMNet looks up the request log with the `HashVal` in the packet. If the request log hits, PMNet forwards this `server-ACK` to the destination (the next PMNet in this route may log the request).

- **Retransmission request from a server (`Retrans`).** In case the server detects any packet loss, it sends a `Retrans` request to the client, going through PMNet. If PMNet has the requested packet logged, PMNet directly sends it to the server and drops this `Retrans` request. Otherwise, PMNet forwards this `Retrans` request directly to the target client.

- **Non-PMNet packets.** As PMNet also serves as a regular network device, it also handles non-PMNet packets by directly forwarding them to the destination.

#### MAT pipeline workflow

PMNet implements the MAT pipeline for packet processing, as shown in Figure 3.8. The pipeline contains three stages: *ingress*, *PM-access*, and *egress*. The *ingress* pipeline first checks whether this packet is a PMNet packet based on its port number in the UDP header (step ❶); non-PMNet packets are directly forwarded to the destination port. Second, for the remaining PMNet packets, it checks the type of the packet based on the `Type` field in the header (step ❷), and forwards `PMNet-ACK` packets to the destination. The *PM-access* stage operates on the request logs. It creates a log upon receiving an `update-req` packet (step ❸), removes it upon a `server-ACK` packet (step ❹), and looks up a log upon a `Retrans` packet (step ❺), by using `HashAddr` as the index. The *egress* stage processes the outgoing packets based on the outcome of the (log) lookup. For `update-req` packets, it first

forwards all of them to the destination server (step ❼); and for those that can be logged in its PM, it additionally generates and sends a `PMNet-ACK` to the client (step ❻). For `Retrans` packets, if the log entry is present, the egress pipeline generates and resends the requested packet to the server; otherwise, it forwards this `Retrans` to the destination client.

During the PM-access stage, PMNet manages log entries in its PM through a DMA engine (i.e., to add/remove/read a log entry). Different from the rest of the MAT pipeline, the PM access stage can suffer from the longer PM access latencies. To prevent blocking incoming packets, PMNet maintains *log queues* (separate queues for reads and writes) that buffer PM access requests (as shown in Figure 3.6). This way, PMNet can handle all incoming packets at line-rate (Section 3.6.2).

### 3.4.3 PMNet Replication

Replication is a common fault-tolerance mechanism that maintains multiple copies of the same data block across various storage servers in a distributed system [145–148]. In case one server fails, the other can be used to recover the corrupted data. To enable fault-tolerance, an update request must commit to all the replication servers. PMNet can accelerate these fault-tolerance systems, further, by replicating data using in-network PMs. Figure 3.9a demonstrates a scheme where the server replicates data. Accordingly, we place two switches in series to maintain two copies of logs as well.

An update request is processed in the following steps: The client sends an update request (step ❶), and the first PMNet switch (#1) logs the request (step ❷) and sends a `PMNet-ACK` (#1) (step ❸). Then, the second PMNet switch (#2) receives the forwarded request, performs logging (step ❹), and sends a second `PMNet-ACK` (#2) (step ❺). On the client side, it continues processing only after it has received *both* `PMNet-ACK` (#1) and (#2). On the server side, the primary server receives the update request and process it (step ❻). Afterward, it sends the updated data to the replication server. Only after replication completes (step ❼), the primary server sends a `server-ACK` to invalidate both logs in the two PMNet switches (step ❽). Even though such in-network replication requires all switches to persist the log prior to sending an acknowledgement, the latencies for persisting data at each switch are overlapped. Figure 3.9b shows the latency benefits of this overlap in the replication procedure for two switches (evaluation in Section 3.6.2).

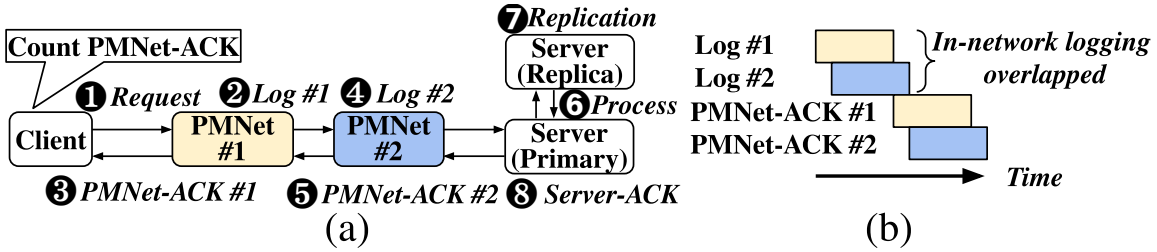



Figure 3.9: (a) A replication scheme with two PMNet switches.
(b) Timeline of in-network replication.

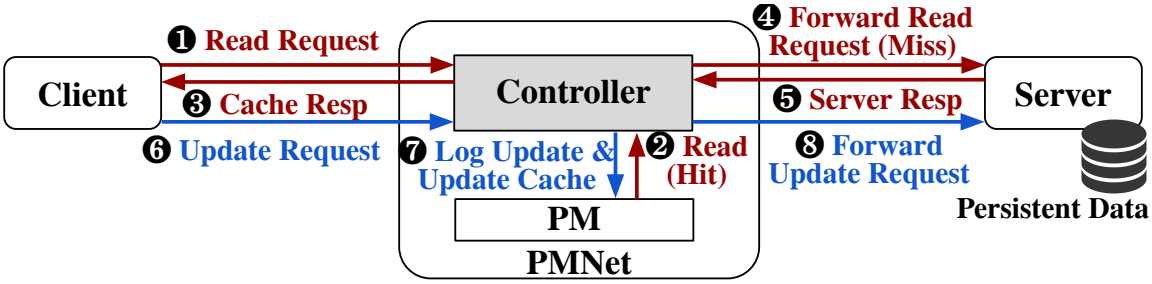



Figure 3.10: Caching steps in a PMNet switch.

### 3.4.4 PMNet Read Caching

Prior works have used programmable switches as a cache for key-value stores [11, 92, 93] but they do not maintain a persistent state of the data cache and cannot mitigate RTT for update requests. To serve both read and update requests, we add a read cache on top of PMNet's persistent log. With read caching enabled, PMNet is implemented as the server's ToR switch, similar to the prior works on in-network read cache, to simplify consistency issues [11]. It maintains a persistent key-value cache that logs update requests and responses to read requests. Figure 3.10 shows this procedure. When a read request arrives at PMNet (step ❶), it looks up the cache. If the request is a hit in the cache (step ❷), PMNet sends a cache-response to serve the read (step ❸). In case of a miss, the read request is forwarded to the server as normal (step ❹), and the response is cached in PMNet when the server replies (step ❺). On the other hand, an update request is first logged in PMNet (step ❼) to move server processing off the critical path (step ❽). In case the key in the update request is a hit to the cache (i.e., has been cached already), it updates the cache to maintain consistency (step ❼).

Figure 3.11 describes a state diagram, where each entry in PMNet has four states: (1) `Invalid`: the entry is empty (initial). (2) `Stale`: the entry is not up-to-date when there is an in-the-flight update to the same key (3) `Persisted`: the request logged by this entry has been persisted on the server, and (4) `Pending`: the request has been logged by PMNet but not persisted by the server. When the state is `Pending` or `Persisted`, the entry can serve for read cache. Here are the state transitions.

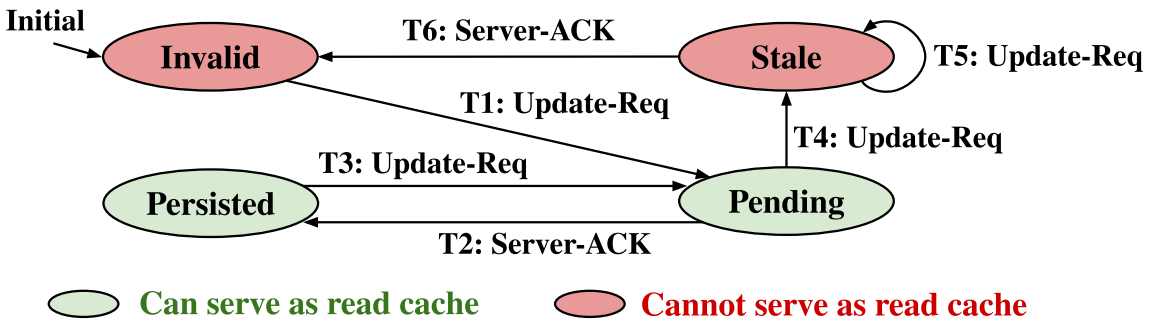



Figure 3.11: A state diagram for PMNet with integrated read cache.

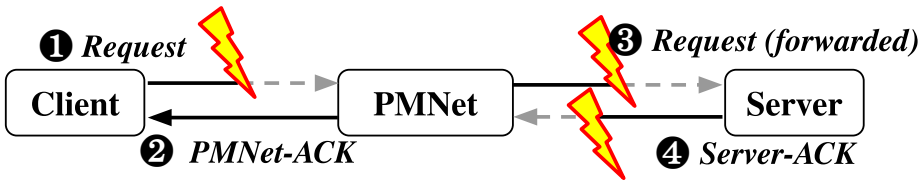



Figure 3.12: Intermittent failure scenarios.

**T1:** Upon receiving an `update-req` from the client, PMNet logs the request. As the server has not persisted it, the state is now `Pending`. **T2:** After receiving a `server-ack`, PMNet is notified that the server has persisted the request. Thus, the status becomes `Persisted`. **T3:** If the same entry (indexed by the key) is updated again (via an `update-req`), PMNet directly bypasses this request and the state goes back to `Pending`. **T4:** When an entry is `Pending` but receives another `update-req`, the state becomes `Stale` as the server will be updated with a new value. **T5:** A `Stale` entry remains `Stale`, after getting another `update-req`. **T6:** Once a `Stale` entry receives a `server-ACK`, it becomes `Invalid` as its prior `update-req` has been persisted.

### 3.4.5 PMNet Failure Recovery

Our work focuses on a system with client-server architecture located in a modern data center. Here, we consider both intermittent failures (such as power outage, software bug and other hardware failures that cause the system to temporarily become unavailable) and permanent hardware failures that are handled via replication (details in Section 3.4.3).

#### Intermittent failures

In this case, all hardware regains their functionality after the failure ends. However, data that is outside the persistent domain is lost. To ensure data integrity, PMNet ensures that accepted requests are safely stored in the persistent domain. We categorize such failures into three cases.

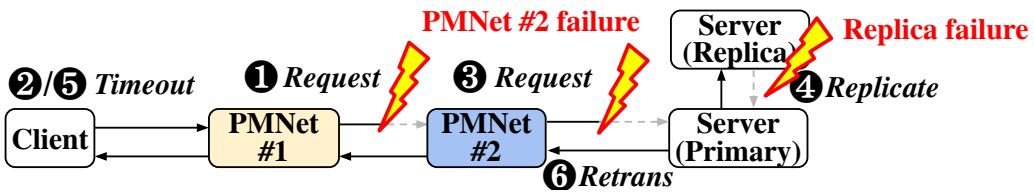



Figure 3.13: Failure-recovery in a PMNet system with replication.

- **PMNet fails before receiving the request from the client** (during step ❶ in Figure 3.12). In this case, as the packet has not been accepted by either PMNet or the server, the client is not acknowledged. Therefore, the client will continue to be stalled by the in-flight request and simply needs to resend the request after timeout (or during recovery).

- **PMNet fails after accepting a request but before a server receives the forwarded request** (during step ❸ in Figure 3.12). We discuss two cases. **(1)** The `PMNet-ACK` was sent to the client before failure (step ❷ is complete). In this scenario, the client has assumed the packet has been processed but the server has not processed it, in fact. During recovery, the server polls PMNet for logged requests with the sequence number starting from the last packet it receives. Then, the server applies logged requests in the same order as they were sent by the client. Note, PMNet itself is agnostic of the packet ordering for better performance, but the server uses `SeqNum` to maintain the order of requests. **(2)** The `PMNet-ACK` was *not* sent to the client before the failure (step ❷ is incomplete). In this scenario, the client has not received the acknowledgement and, therefore, it will be stalling on the current request. After the client times out, it will resend the request again.

- **PMNet fails before receiving the `server-ACK`** (during step ❹ in Figure 3.12). PMNet first resends the logged requests to the server. Upon reception of the resent request packets, the server checks the `SeqNum` of each packet. As the server has processed the request, the latest `SeqNum` is greater than that in the resent request and, therefore, the server drops these requests and sends a make-up `server-ACK` to invalidate the log entries in PMNet.

### Permanent failures

In this type of failure, the hardware that stores persistent data (PMNet or server) cannot be recovered. Such failures are handled by replicating data in the persistent domain across multiple devices. We further categorized permanent failures into three cases.

- **A PMNet fails permanently after accepting a request but before the server receives the forwarded request.** For PMNet to handle permanent failure, the update request must persist in all PMNet all the way to the server before it is accepted (as in Section 3.4.3). We discuss two cases. **(1) PMNet #2 fails before both PMNet devices have accepted the request** (step ❶ in Figure 3.13). Because the client has not received both `PMNet-ACK`'s to satisfy the replication requirement, it will be stalling on the current request until it times out[2] and resends the request (step ❷ in Figure 3.13). **(2) PMNet #2 fails after both PMNet devices have accepted the request and send `PMNet-ACK`** (step ❸ in Figure 3.13). During recovery, the server polls PMNet for the logged requests. Because all PMNet devices have logged the request, any surviving PMNet can retransmit the request to the server.

- **Server replication system fails before sending `server-ACK`.** In a replication system, the primary server sends `Server-ACK` only after it has committed the update in all replicas. Thus, when a replication server fails (step ❹ in Figure 3.13), the server will not send `server-ACK` to invalidate the logged requests in PMNet. Eventually, when PMNet's log entries are full, PMNet forwards newer requests directly to the server. However, those requests will not be processed by the failing replica and the client will eventually timeout[2] (step ❺ in Figure 3.13). All the in-flight requests are now logged in PMNet with equivalent replication strength (the same number of PMNet devices as the replicas). Once the replication system is up, the primary server sends `Retrans` (step ❻ in Figure 3.13) to get the logged requests from PMNet. And, the retransmitted requests will be processed by the server in-order based on the sequence number. Note that these systems typically monitor servers' status using heartbeats. Therefore, the client will be notified as soon as a replica fails—the client would not need to wait until it times out.

#### Failure of other components

When components outside of the persistence domain fail, such as the client or other non-persistent network devices, the system follows the original procedure for recovery. The recovery procedure does not change as the persistence guarantee of those devices remains the same.

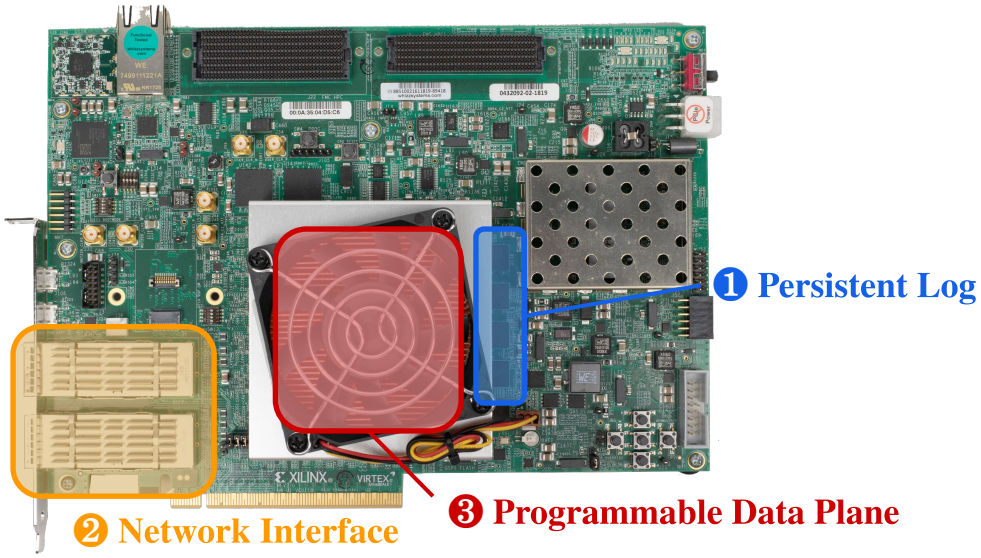



Figure 3.14: FPGA platform for PMNet implementation.

## 3.5 PMNet Implementation

### 3.5.1 Hardware Implementation

We express PMNet's processing pipeline in the P4 language [149]. We choose Xilinx UltraScale+ VCU118 platform (Figure 3.14) as the hardware for the programmable network device and maintain the request logs in its 2GB on-board DRAM (labeled as component ❶). The DRAM write latency in the FPGA is 273 ns (due the slow DMA engine on the FPGA) which is close to Optane PM's write latency [150]. We adapt the open-source code from NetFPGA-SUME (with 10G Ethernet MAC) [151] to our UltraScale+ platform and integrate PMNet into it. The network interface is labeled as component ❷. The whole design uses 13% of LUT, 19% of BRAM, and 31% of IO resources of the FPGA. The FPGA chip is labeled as component ❸. We use a Li-ion battery module to back the device during power failure.

**Design choices** The in-network PM needs to log all the in-flight update requests. In our evaluation, the 99th-percentile RTT of update requests is 350 µs. If we conservatively take 500 µs as the maximum RTT and 10 Gbps as the bandwidth, the bandwidth-delay product (BDP) is:

$$BDP_{Net} = RTT \times BW = 500 \cdot 10^{-6} \times 10 \cdot 10^{9} \approx 5 Mbits. \tag{3.1}$$

[2]The timeout value is 2× 99th-percentile RTT, which is 700 µs.

Table 3.1: PMNet software interface.

| Client Software Interface | |
|---|---|
| `PMNet_send_update()` | Send an `update-req` to server |
| `PMNet_bypass()` | Send a `bypass-req` to server |
| `PMNet_start_session()` | Start a session |
| `PMNet_end_session()` | End a session |
| **Server Software Interface** | |
| `PMNet_recv()` | Receive requests from client |
| `PMNet_ack()` | Send ACK to PMNet |

Table 3.2: System configuration.

| Server Configuration | |
|---|---|
| CPU | Intel Cascade Lake, 2.1GHz, 20 cores |
| DRAM | 6×32GB DDR4, 2666MT/s |
| PM | 2×128GB Intel DCPMM, Interleaved, 2-1-1 Config, App Direct Mode, Mounted as EXT4-DAX |
| NIC | Mellanox ConnectX-3 MCX314A |
| **Client Configuration** | |
| CPU | Intel Haswell, 3.6GHz, 6 cores |
| DRAM | 4×16GB DDR4, 2133MT/s |
| NIC | Mellanox ConnectX-3 MCX314A |
| **Software System** | |
| OS | Ubuntu 19.10, Linux kernel v5.3.0 |
| Tools & Libs | gcc/g++-9.2, PMDK-1.8, daxctl/ndctl-65 (server only) |

Our FPGA board has sufficient memory capacity to log all these on-going update requests. Because of the slower PM access latency, we buffer the accesses to the in-network PM on the PMNet device (switch or NIC) to process packets at a line rate. The required buffer size also follows a BDP calculation where the delay equals to the memory-access latency.

$$BDP_{PM} = PMLatency \times BW = 100 \cdot 10^{-9} \times 10 \cdot 10^{9} \approx 1kbits. \tag{3.2}$$

We conservatively use 4 KB of SRAM as the queue size for logging both reads and writes to PM. Section 3.7 discusses support for even higher network bandwidth with PMNet.

### 3.5.2 Software Implementation

We develop an easy-to-use software interface that allows programmers to adapt existing workloads to a PMNet system (`PMNet_interface.h`). Table 3.1 lists the interface functions. Programmers need to overwrite the existing send and receive functions of the system's socket interface with the PMNet version. Then, the PMNet library operates on these functions and encapsulates payload in PMNet-compatible formats. In addition, PMNet library serves two major purposes. First, it accepts incoming packets from PMNet to mitigate the RTT for the client (Section 3.4.2). Second, it maintains the ordering and integrity of PMNet packets for the server (Section 3.4.1).

## 3.6 Evaluation

### 3.6.1 Methodology

#### System setup

We evaluate PMNet using a testbed, described in Table 3.2. The server is equipped with Intel's DC Persistent Memory [3] that are mounted in DAX-FS mode for workloads to directly manage PM. The clients contain normal DRAMs on the machines and send requests to the server for persistent data access. Both the server and clients use Mellanox NICs for network connection. In total, we have 4 client machines, each running up to 16 client instances (64 in total), and 3 Xilinx UltraScale+ FPGAs that are programmed as PMNet-NICs/Switches. In the PMNet-Switch configuration, the client machines are connected to a top-of-the-rack PMNet switch (Section 3.5.1). Due to the limited number of Ethernet ports on the FPGA board, we place a regular switch (with sub-microsecond latency) in the middle of clients and the FPGA to merge their traffic. In the PMNet-NIC configuration, the client machines are connected to a regular switch directly; for the server, the FPGA is placed as a bump-in-the-wire between the server's NIC and the ToR switch, similar to recent Microsoft's SmartNIC setup [99, 152].

#### Evaluated workloads

We evaluate workloads from Intel's PMDK library [121] and a PM-optimized version of Redis from Intel [110]. We use a YCSB-like client [153] to generate and send read/update requests to the server. We also evaluate the performance of PMNet with two real workloads: a Twitter workload based on the Twitter Clone tutorial [1] and the online transaction processing benchmark, TPCC [2]. All server workloads manage persistent data in PM directly through the DAX-FS support. To support these workloads, we modified 11 and 7 lines of code in Redis and PMDK workloads, respectively. The payload of each read/update request is 100 Bytes, by default, unless specified otherwise. During evaluation, we skip the first 10k (warm-up) requests for more precise results.

#### Baseline protocol

We implement the driver program for PMDK workloads (B-Tree, C-Tree, RB-Tree, Hashmap, and Skip List) using UDP. Therefore, both the baseline and PMNet of these workloads use UDP. Redis, Twitter, and TPCC are originally based on TCP. Although UDP is faster, adapting them to UDP introduces a 9% slowdown due to the conversion overhead. Thus, we keep the original TCP-based

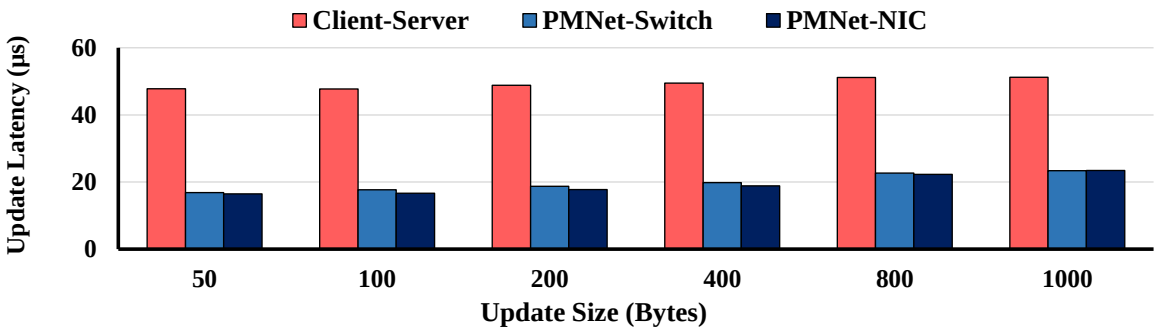



Figure 3.15: Update latency of an ideal request handler with variable request sizes.

communication as the baseline. This way, all the baselines are evaluated with their *best-performing* protocols.

#### Design points

We test PMNet under three system configurations for comparison.

- **PMNet-Switch:** A system with PMNet in the ToR switch of the server rack.
- **PMNet-NIC:** A system with PMNet as server's NIC.
- **Client-Server:** A baseline system which forwards all network packets to the destination.

### 3.6.2 Evaluation Results

#### Latency of microbenchmarks

We start with evaluating the raw latency and bandwidth of PMNet. In a practical system, the server-processing time can be the bottleneck. Therefore, we implement a microbenchmark with an *ideal request handler* on the server-side that acknowledges the client upon reception of the request, without processing it. Hence, the network latency becomes the primary bottleneck.

We first evaluate the latency benefit of PMNet. Figure 3.15 shows the round-trip time latency of PMNet-Switch and PMNet-NIC as we vary the payload size from 50B to 1000B, using a single client. We make two observations from this figure. First, PMNet-Switch and PMNet-NIC provide 2.83× and 2.90× speedup, respectively, over the Client-Server with a payload size of 50B. However, the benefit decreases with larger payloads as the payload processing time goes up in the network device. For example, both PMNet-Switch and PMNet-NIC provide around 2.19× speedup compared to the Client-Server with 1000B payloads. Second, we observe that the difference in absolute latency

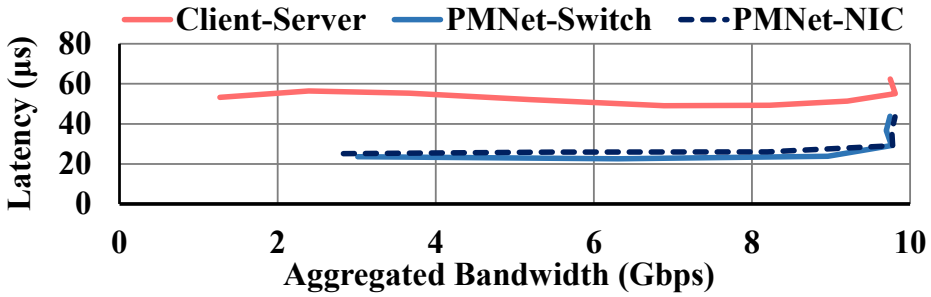



Figure 3.16: Bandwidth vs. latency under stress test.

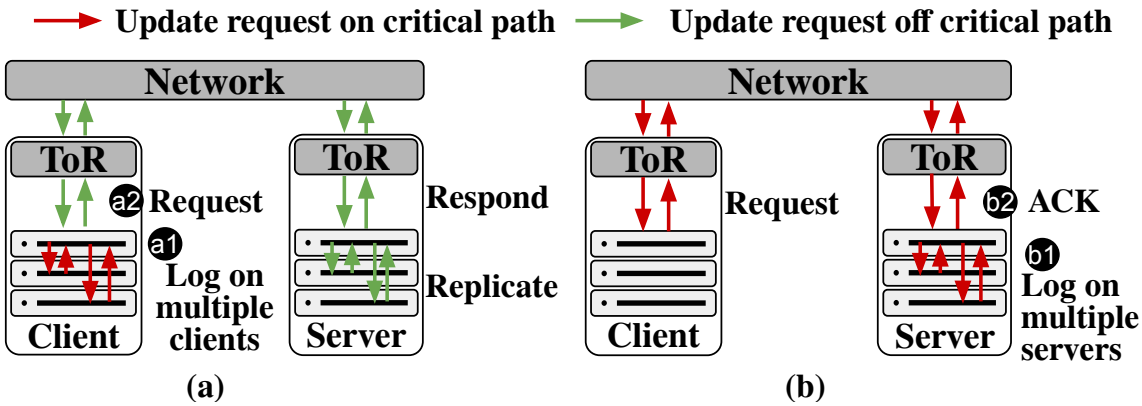



Figure 3.17: Alternative designs: (a) client-side logging with replication and (b) server-side logging with replication.

between PMNet-Switch and PMNet-NIC is almost negligible (under 1 µs) as wire latency is low and the most benefit comes from moving the server processing and network latency off the critical path.

Next, we stress test the bandwidth using an ideal server-side request handler. On the client-side, we scale the number of client instances and keep sending 1000B requests to the server to saturate the bandwidth. Figure 3.16 displays stress testing results. First, both PMNet configurations and the Client-Server follow a similar trend where the latency remains the same when the total bandwidth is low, and there is a spike in latency when the bandwidth reaches the physical limit at 10 Gbps. Second, when the bandwidth is less than 10 Gbps, both PMNet-Switch and PMNet-NIC consistently have better latency than the Client-Server as they move the server off the critical path. As both PMNet configurations are equally effective in terms of update request response time and maximum bandwidth, in the next sections, we discuss PMNet performance using switches only.

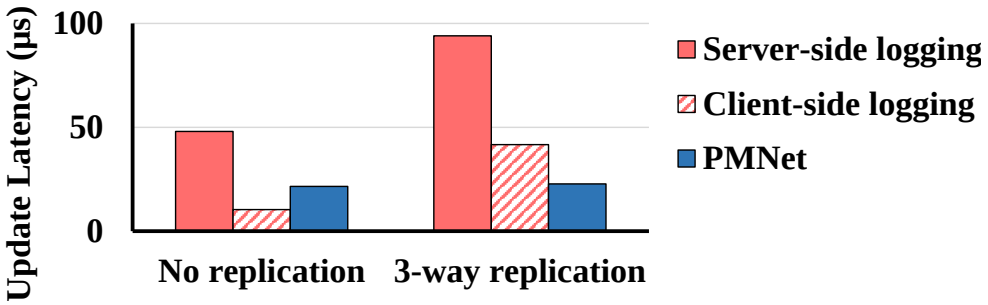



Figure 3.18: PMNet vs. alternative designs: server-side logging and client-side logging.

**Comparison with alternative designs**

We compare PMNet with two alternative designs (request payload is 100B). To maximize the performance difference due to communication, we use the aforementioned microbenchmark (Section 3.6.2).

**Client-side logging** (Figure 3.17a) locally logs the request and then lets the client proceeds (step a1). The client then forwards the request (step a2), such that the server's network stack and processing time are off the critical path. We implement client-side logging in a separate, dedicated software process, following a client-side persistent logging (caching) design [154]. The application directly overwrites the original socket interface to send logs to the client-side logger.

**Server-side logging** (Figure 3.17b) logs the request on the server upon reception (step b1) and immediately notifies the client (step b2), in order to move the server's processing time off the critical path. We implement the server-side logging following a persistent write logging (caching) design [155].

Figure 3.18 compares the latencies in these two designs. First, without replication, client-side logging is faster than PMNet as it does not go through the client's network stack (10.4 µs compared to PMNet's 21.5 µs). Whereas, server-side logging is much slower than PMNet because a large fraction of the server network latency remains on the critical path (47.97 µs). Second, with 3-way replication enabled, client-side logging becomes inefficient (41.61 µs) because it needs to communicate with other clients to replicate the logs. Similarly, server-side logging latency also increases significantly due to communication (94.02 µs). In comparison, PMNet consistently performs well even with replication (22.8 µs and 21.5 µs with and without replication), as the communication latency among replicas is off the critical path.

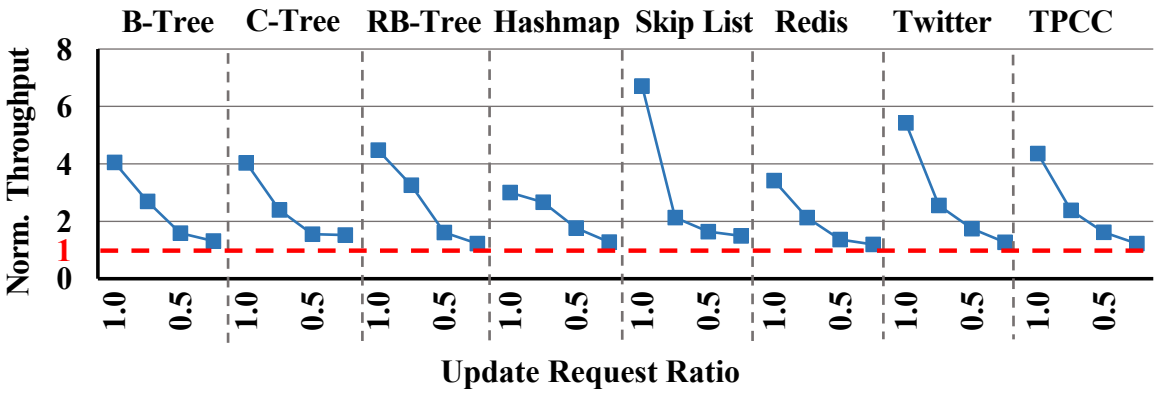



Figure 3.19: Throughput normalized to Client-Server with variable update/read ratio.

### Application performance

Prior works have shown that the update/read ratio varies [142,156]. Figure 3.19 shows the normalized throughput in real-world applications with PMNet, when we vary the update ratio from 100% to 25%. We make two observations: First, PMNet provides, on average, 4.31× speedup over the Client-Server with 100% update requests. Second, as the ratio of read requests increases, the throughput improvements from PMNet decrease. This trend is expected as PMNet focuses on update requests. We show the performance benefit of PMNet with read caching in the next section.

### PMNet with read caching

Our read-cache implementation is based on "key" lookups using the GET/SET interface in the key-value store workloads. As a result, this experiment only includes the key-value store based workloads from PMDK [121] and Redis [110], and excludes workloads with complex queries, such as Twitter [1] and TPCC [2]. Figure 3.20a and 3.20b show the cumulative distribution function (CDF) of the latency with 100% and 50% update requests, respectively. We make three observations from this figure. First, the average latency of PMNet with caching is 3.36× lower than the Client-Server system. Second, when 50% of requests are updates, the latency of PMNet without caching has a noticeable transition point at the 50th-percentile (blue lines in Figure 3.20b), where latencies become close to the Client-Server system afterward. This happens because only half of the requests are updates and have been optimized by PMNet. In comparison, when 100% of the requests are updates (Figure 3.20a), the latency does not drop and provides 3.23× better tail latency than the Client-Server design. On the other hand, when caching is integrated into PMNet, the latency benefit does not stop at 50th-percentile but keeps continuing (green lines in Figure 3.20b)—as it serves *all* update requests and *most* read requests (cache hits) with a sub-RTT latency. Third, workloads with

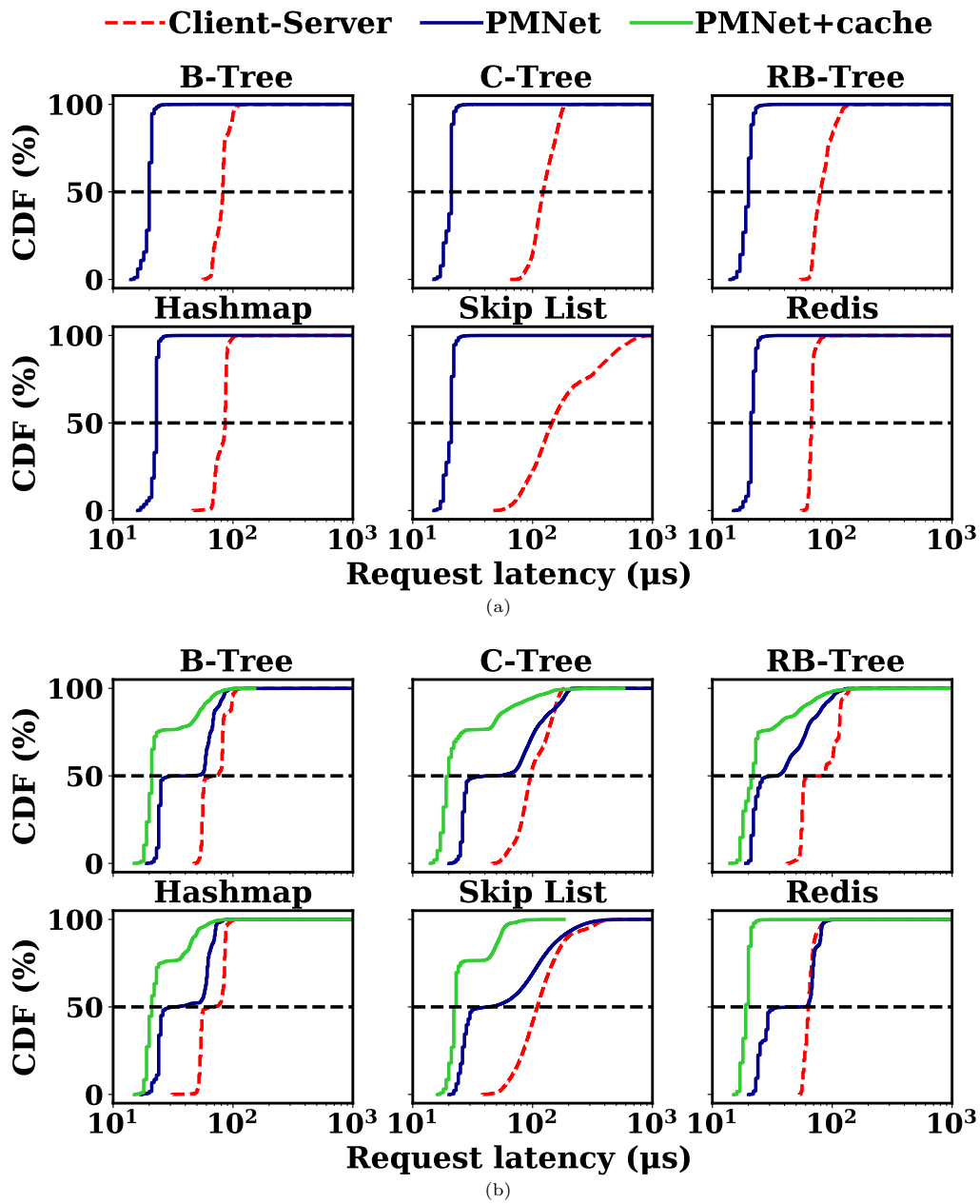



Figure 3.20: CDF of request latency with (a) 100% and (b) 50% update request.

a higher hit rate (e.g., Redis) significantly benefit from the 99th-percentile latency with caching. We conclude that, with the integration of read caching, PMNet effectively reduces the latency of both update and read requests.

### PMNet with replication

The replication scheme connects three PMNet switches in series to implement a 3-way replication in the network. Figure 3.21 shows the benefit of in-network replication compared to a Client-Server

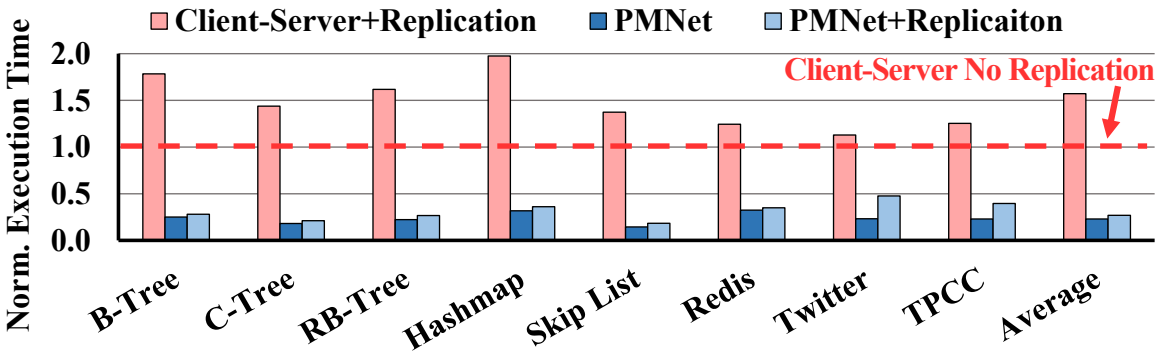



Figure 3.21: Update request latency in a 3-way replication system (normalized to the no-replication Client-Server design).


Figure 3.22: Update throughput with optimized network stack.

system that performs replication on the server-side. We make two observations. First, PMNet with replication provides 5.88× better performance than server-side replication on average. Second, the overhead due to in-switch replication is low, as the latency of persisting logs is overlapped in our mechanism. The 3-way replication introduces 16% overhead over a PMNet system that only logs the updates once.

### Recovering from server failures

We evaluate how a PMNet system recovers from server failures. To mimic failure, we manually cut off the power of the server and let PMNet resend logged requests after power has been restored. For each workload, we saturate the network bandwidth to create the worst-case scenario where PMNet has the maximum number of requests logged. On average, it takes 67 µs to resend a single request and 4.4 seconds to resend all pending requests in the log. Even in the worst case, the entire recovery procedure (resend + application recovery) only takes 9.3 seconds which is a small fraction of the server's 2∼3-minute boot-up time [157].

**PMNet with an optimized network stack**

The latency breakdown in Figure 3.2 (Section 3.2.3) shows the client- and server-side latency consists of both network stack and processing time. With an optimized network stack, the network stack overhead can be significantly reduced. In this experiment, we use libVMA [139] to reduce the network stack time by moving network procedures into the user-space and avoiding the expensive context switching from the kernel. To evaluate an ideal scenario where the server-side overhead is minimal, we use the microbenchmark introduced in Section 3.6.2. And, both the client and the server are running optimized network stacks. Figure 3.22 compares the update-throughput among four designs: Client-Server, PMNet, Client-Server + libVMA, and PMNet + libVMA. The result shows that, without libVMA, PMNet provides 3.08× better throughput. After applying libVMA, the server network stack overhead is significantly reduced. Nonetheless, the integration of PMNet still provides 3.56× better throughput. Although the speedup is lower—part of the server-side overhead has been reduced by libVMA—the benefit from PMNet is still significant as PMNet moves the remaining server processing time off the critical path.

## 3.7 Discussion

In this section, we discuss the network bandwidth and PM performance, and alternatives to PMNet.

**Reaching Higher Network Bandwidths** Fundamentally, PMNet supports higher bandwidths. First, the relatively slower PM access is decoupled from the network traffic by queuing the PM access in log queues. By increasing the log queue size according to the bandwidth-delay product (BDP) (Equation 3.1) of a high-bandwidth network, PMNet can buffer incomplete accesses to PM. To support a 100 Gbps network, only a 10 kbit (or 1.25 kB) log queue buffer would suffice. Second, the PMNet only needs to buffer ongoing update requests that have not been committed to the server. Only 500 Mbit (or 62.5 MB) of PM is needed to buffer the in-flight update requests in a 100 Gbps network (Equation 3.2).

**PM Write Bandwidth** In our implementation, we use battery-backed DRAM on the FPGA board, that has a bandwidth of 2.5 GB/s, similar to the per-DIMM bandwidth of Intel's Optane PM. We expect that future PM technologies will enable much higher bandwidth, such as the higher-bandwidth battery-backed NVDIMMs [103], the emerging persistent cache [158], and alternative PM media (e.g.,

STT-RAM [159], ReRAM [160]). With a higher PM bandwidth, PMNet can handle higher update request bandwidth.

**External Persistent Storage** We integrated PM into a network device. Alternatively, other types of storage can also maintain persistent data. For example, switches can access a network-attached PM device [161] (or SSD [162]) instead of keeping persistent data on-board. However, such designs add additional network latency to persist data in the network device, eventually inflating the critical path of client execution. Further, as our BDP calculations have shown (Section 3.5.1), the PM capacity requirement is relatively small, and therefore, it is unnecessary to use an external device for persisting requests.

# Chapter 4

# Optimizing Data-Intensive ML for edge systems

## 4.1 Introduction

Large Language Models (LLMs) enable new applications such as smart assistants [163, 164]. The processing of these powerful LLMs is usually offloaded to the datacenter due to the enormous resources required. However, the latest mobile platforms enable smaller LLM to run locally. These lightweight models cannot directly compare with the state-of-the-art hundred-billion parameter LLMs. A promising solution is to build a compounding system–by integrating LLMs with Retrieval Augmented Generation (RAG) [4], allowing these smaller models to leverage local personal data, thereby enhancing their ability to generate high-quality responses.

Even though RAG removes the requirement for a heavy-weight LLM for generation, retrieval still has a high overhead. The core of a RAG system is a vector embedding database that enables vector similarity search. Unlike LLMs, the overhead of RAG mainly comes from its memory footprint. For example, a Flat Index stores and sequentially searches every vector representation of the data chunks to identify the closest match to the query. For example, in the fever [165] dataset, a vector database that holds 5.23 million records has an index size of 18.5 GB. In comparison, mobile devices usually have around 4–16 GB of main memory [166]. Table 4.1 shows examples of mobile devices. Thus, even the whole memory on a mobile platform is not sufficient to run a large vector database. On

Table 4.1: Edge system comparison

| System | Memory | Compute Units |
|---|---|---|
| iPhone 16 Pro [167] | 8 GB | CPU+GPU+NPU |
| Galaxy S24 [168] | 8 GB | CPU+GPU+NPU |
| Jetson Orin Nano [169] | 8 GB | CPU+GPU+TensorCore |
| L40 [170] (Server) | 48 GB | GPU |

the other hand, storing the vector database on disk introduces substantial access latency, impacting performance.

This work focuses on the challenges of implementing Retrieval Augmented Generation (RAG) on edge systems. We find that naively keeping the entire index in the main memory would not fit into the memory of mobile platforms. A Flat index that performs a sequential search of all embeddings is not only expensive in terms of computation but also trashes memory leading to poor performance. In contrast, Two-level Inverted File (IVF) index clusters embeddings of data chunks into clusters. The retrieval process first searches for the closest centroid and then performs a second search within the cluster, avoiding an expensive sequential search of all embeddings. However, keeping all embeddings in memory still leads to excessive memory thrashing and increased latency. Our solution is to keep the first-level centroid in memory and generate the second level online. Through profiling the RAG on mobile edge platforms using widely-used RAG benchmarks [171], we find that both the data access pattern and the access latency are highly skewed. First, most of the embeddings are not searched during the retrieval process. Second, the cost of generating the embedding is not the same for all clusters and has an extreme tail distribution. These skewness leave space for further optimizations.

In this work, we develop a system that enables RAG for edge platforms, by fitting the vector database in the limited edge memory while ensuring that the response time meets the service level objectives (SLOs) of mobile AI assistant applications. Based on these observations, our key ideas are the following: First, we prune the vector embedding of the data embeddings within centroid clusters which are only used for second-level search to save the memory capacity. EdgeRAG then generates the embedding online during the retrieval process. However, due to limited computing on edge systems, generating embedding online could suffer from long embedding generation latency from large tail clusters. To overcome this challenge, our second solution is to pre-compute and store the embeddings of large tail clusters to avoid long tail latency of generating embeddings of data within those tail clusters. Then, EdgeRAG can adaptively cache the remaining embeddings to minimize

redundant computation and improve overall latency, based on the spare memory capacity and SLO requirements.

We evaluate EdgeRAG on an edge platform based on NVIDIA Jetson Orin Nano equipped with 8 GB of shared main memory, similar to a mobile platform with neural processing capabilities [168]. We use 6 workloads from the BEIR benchmark suite [171]. Because EdgeRAG is based on the IVF index which trades off accuracy for performance compared to the Flat index baseline, we tune the retrieval hyperparameters to normalize the recall to maintain the same generation quality. We also evaluate the generation quality using GPT-4o [172] as an LLM evaluator [173]. We use the time-to-first-token (TTFT) latency as the main metric. The result shows that EdgeRAG offers 1.8 × faster TTFT over the baseline IVF index on average and 3.82 × for larger datasets. At the same time, EdgeRAG maintains a similar generation quality with recall and generation scores within 5 percent of the Flat Index baseline while allowing all of our evaluated datasets to fit into the memory and avoid memory thrashing. We also show that with a relaxed latency SLO, EdgeRAG can handle embedding databases up to 1.5 TB in size with 8 GB of memory.

In summary, the contributions of this work are the following:

- We identify two key challenges of implementing RAG on edge devices: First, limited memory capacity does not allow loading of large vector embedding database in the memory leading to memory thrashing and poor performance. Second, limited computing power of edge devices which slows down online embedding generation, especially on few large tail and repeatedly used clusters.

- To enable scalable and memory-efficient RAG on edge systems, we develop EdgeRAG which improves upon the IVF index by pruning second-level embeddings to reduce memory footprint and generate the embedding online during retrieval time. EdgeRAG mitigates long tail latency from generating the embeddings of tail cluster by pre-computing and storing those tails. To further optimize latency, EdgeRAG selectively caches generated embeddings to reduce redundant computation while minimizing memory overhead.

- We implement EdgeRAG on the Jetson Orin edge platform and evaluate our system with 6 datasets from the BEIR benchmark. The results show that EdgeRAG significantly improves the retrieval latency of large datasets with embedding sizes 2.31× the memory capacity with only a slight reduction in retrieval and generation quality.

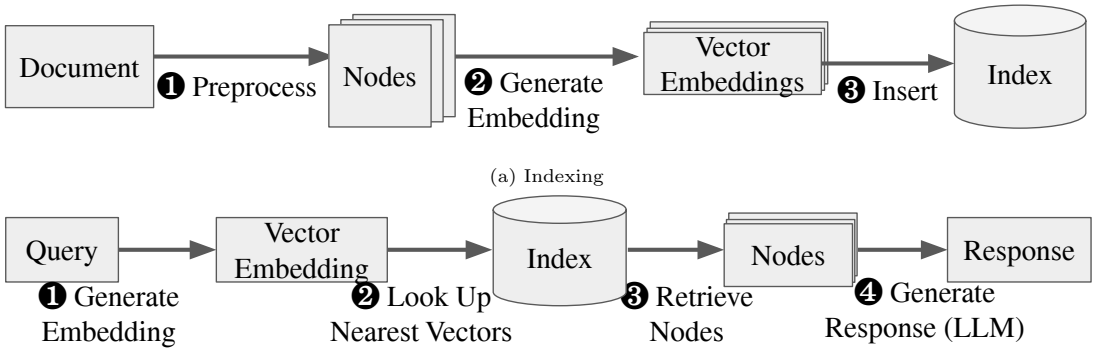
## 4.2 Background

### 4.2.1 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) [4] is a technique used to improve the accuracy of the Large Language Model (LLM) by integrating an external knowledge base with the LLM. Instead of retraining the entire LLM with new information, RAG adds new data to the knowledge base. The RAG consists of two parts: Indexing and Lookup.

**Indexing:** Adding new data to the database involves a process called indexing, shown in Figure 4.1a. This step involves dividing the data into smaller, overlapping chunks represented by Nodes (step 1). Each chunk of data is then fed into an embedding model, which generates a unique high-dimension vector representation for that particular piece of data (step 2). Finally, these embedded vectors, which act as compressed representations of the data, are stored in a separate database for efficient retrieval later (step 3).

**Lookup:** When a user submits a request, the system performs a lookup process as shown in Figure 4.1b. First, the user's request goes through the same embedding model as the data in an indexing phase (step 1). This creates a vector representing the user's query. Then, this embedding of the query is compared to the embedded vectors stored in the dedicated database. This is also known as vector similarity search (step 2). The system returns the closest matches among these stored indexes (step 3). Finally, the data nodes associated with the closest matching indexes are retrieved and fed to the large language model (LLM) to generate a response (step 4). Essentially, the lookup process helps retrieve data in the database related to the user's query, which LLM then uses to generate a response.




(a) Indexing

(b) Lookup

Figure 4.1: RAG Pipelines.

### 4.2.2 Vector Similarity Search

As mentioned in the previous section, RAG searches for the embedding representation of the data chunk in the database with the closest distance to the query embeddings. However, embeddings often reside in high-dimensional spaces. Techniques like sorting, which are efficient in low-dimensional spaces, become less effective or even infeasible in high-dimensional spaces. Consequently, finding similar embeddings in high-dimensional spaces requires computationally expensive operations. Distance metrics such as Euclidean distance or cosine similarity involve calculations across all dimensions of the vectors, making the process time-consuming.

### 4.2.3 RAG Indexing Methods

We previously discussed the challenges of vector similarity search. Now, we'll explore indexing techniques that can significantly improve search latency.

**Flat Index** compares a query embedding to every embedding in the index. While this method can achieve a high accuracy, it can be computationally expensive, especially for large datasets.

**Inverted File (IVF) Index** [174] employs clustering algorithms to group similar documents into clusters. Figure 4.2 illustrates IVF's retrieval procedure. During query processing, the system first compares the query to the centroids of these clusters in the first level index (step ❶). This initial comparison significantly reduces the number of documents to be examined, leading to faster search times. Then the system searches the second level index for the most similar embedding (step ❷, ❸). Finally, the system retrieves data associated with the embedding (step ❹). However, this approach can potentially miss relevant documents that don't align perfectly with cluster boundaries. To mitigate this, the system may also need to search adjacent clusters, which can slightly increase search time but improve accuracy.

## 4.3 Motivation

### 4.3.1 Memory Limitation on Edge Platforms

Figure 4.3 presents the latency breakdown of the RAG system on an edge platform. End-to-end latency is categorized into three phases: 1) Retrieval latency, the duration of vector similarity search to identify relevant text chunks; 2) First Token Latency or Prefill, the time from model input (query and retrieved chunks) to the generation of the first output token; and 3) Generation latency, the

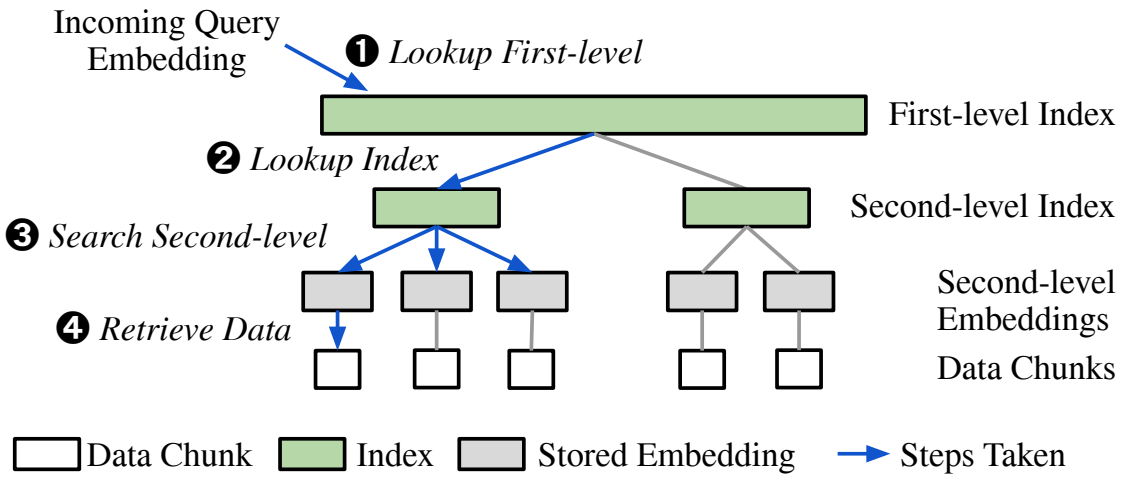



Figure 4.2: Retrieval process of Inverted File Index

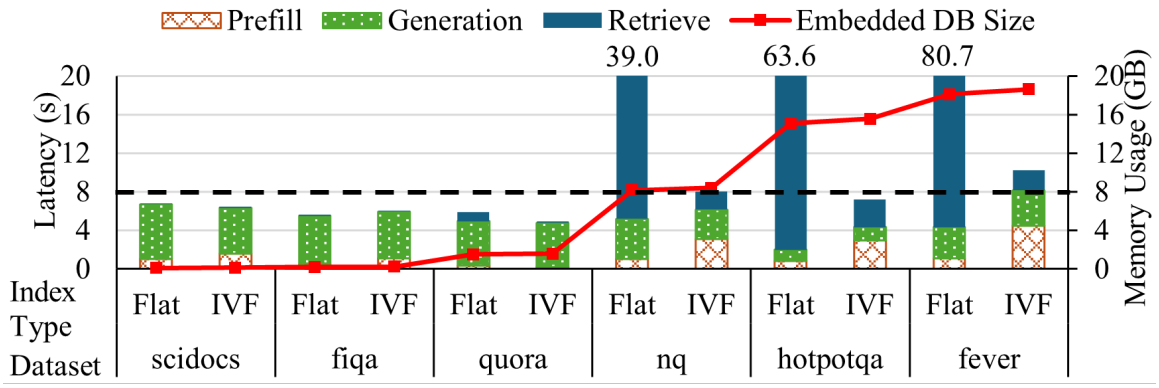



Figure 4.3: RAG latency breakdown and embedded database size

time from the first to the last generated token. The combined Retrieval and First Token latency, or Time-to-First-Token (TTFT), directly affects user-perceived latency, as it represents the delay between query submission and the display of the initial response. While IVF index exhibits slower latency growth compared to Flat index, both configurations are susceptible to memory thrashing for datasets exceeding available memory (nq, hotpotqa, fever). This results in significant latency penalties as the system is forced to repeatedly load and unload the embedding database and the model from the storage. Thus, efficient memory utilization is critical to mitigate this issue.

### 4.3.2 Compute vs. Data Movement trade-offs

One way to reduce the memory footprint of embeddings is to generate them only when needed for incoming queries. Because the two-level IVF index primarily searches embeddings linked to a small subset of first-level centroids, most of these embeddings can be generated during the retrieval process itself. This strategy avoids the significant memory overhead of storing all embeddings. As shown in

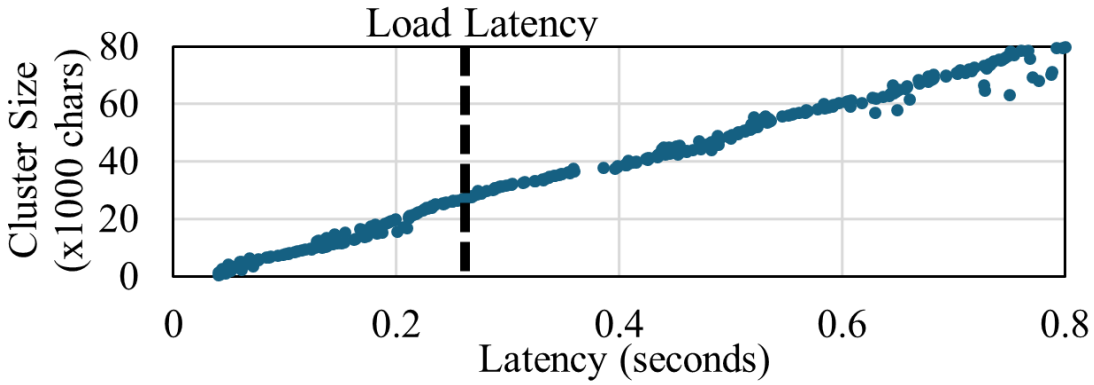



Figure 4.4: Embedding Generation Rate of different cluster size

Figure 4.4, generating embeddings for clusters smaller than 24000 characters or approximately 8000 tokens is faster than loading them from the storage. This indicates that online embeddings generation can not only save memory but also potentially improve retrieval latency. However, generating embedding for large cluster could also cause long tail latency.

## 4.4 High-level Ideas

We present EdgeRAG, a memory-efficient RAG system by selectively pruning second-level embeddings. Upon receiving an incoming query, EdgeRAG generates related second-level embeddings during runtime. However, online embedding generation presents several key challenges: (1) High Latency Variability: Embedding generation time is influenced by data chunk size, leading to unpredictable latency. (2) High Computational Cost: Generating embeddings is computationally expensive. Next, we describe the high-level ideas of EdgeRAG in this section.

### 4.4.1 Selective Index Storage

Pruning second-level embeddings reduces memory footprint, but shifts embedding generation from indexing to retrieval. This can increase retrieval latency if generating embeddings takes longer than accessing pre-computed ones. We make a key observation about embedding generation.

Figure 4.5 presents the distribution of embedding generation times for various clusters within the nq datasets. The majority of clusters exhibit generation latencies under 500 milliseconds. Nevertheless, a subset of clusters, albeit infrequent, may experience generation times exceeding 2 seconds. This distribution highlights a “tail-heavy” characteristic, where a small proportion of clusters significantly impact the overall retrieval time.

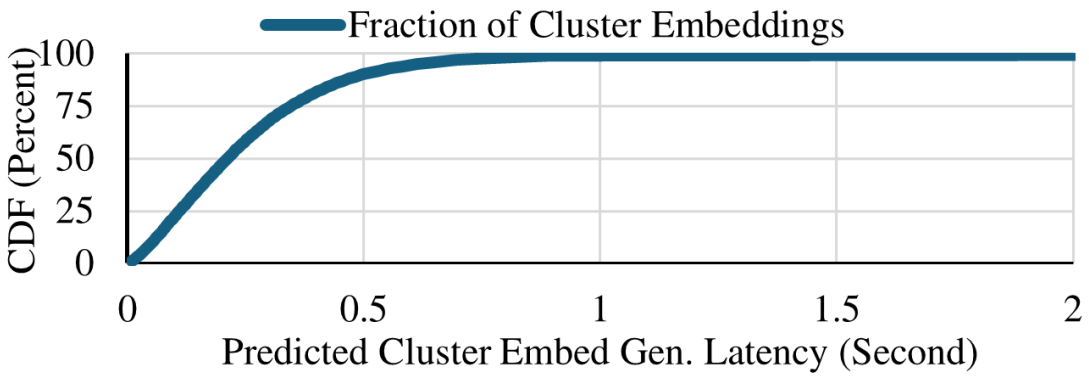



Figure 4.5: Cluster Embedding Generation Cost of nq dataset

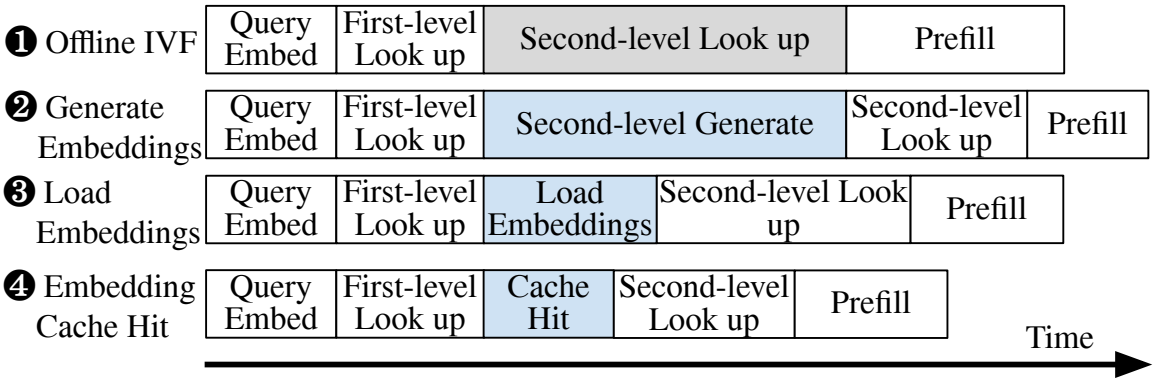



Figure 4.6: Time-to-First-Token (TTFT) breakdown of EdgeRAG Retrieval Process.

To mitigate long-tail latency in large clusters, EdgeRAG employs a hybrid approach. During the initial indexing phase, clusters are profiled to estimate embedding generation latency. Clusters exceeding the latency threshold are identified and their embeddings are precomputed and stored. At query time, embeddings are retrieved from storage if available (❸ in Figure 4.6), bypassing the long latency of online embeddings generation phase (❷ in Figure 4.6). For clusters not precomputed, embeddings are generated in flight. Algorithm 1 illustrates our Selective Index Storage.

### 4.4.2 Adaptive Cost-Aware Caching

Beyond the latency challenges presented by generating embeddings for large clusters, we observed frequent reuse of embeddings for smaller, more common clusters. An analysis of queries across various datasets revealed a substantial degree of overlap in the accessed clusters. This overlap is quantified in Table 4.2 through the chunk reuse ratio. The significant reuse observed across all datasets implies that embeddings for clusters associated with these frequently accessed data chunks must be repeatedly generated. Therefore, EdgeRAG also caches generated embeddings to avoid

**Algorithm 1** Selective Index Storage

```
DataEmbeddings = embed(Datachunks)
Centroids = cluster(Embeddings)
for DataEmbedding in DataEmbeddings do
  Centroid = Centroids.search(DataEmbedding)
  Centroid.add(DataEmbedding)
end for
for Centroid in Centroids do
  // Compute Cluster's Embedding Gen. Latency
  Centroid.GenLatency
  =SUM(len(Centroid.Datachunks))/GenRate
  if Centroid.GenLatency > SLO then
    Centroid.saveEmbeddings()
  end if
end for
```

Table 4.2: Evaluated datasets.

| Dataset | Corpus | # Records | Embeddings | Unique Access | Total Access | Reuse Ratio | Fit in Dev. Mem |
|---|---|---|---|---|---|---|---|
| scidocs [175] | 86 MB | 3.6 k | 113 MB | 1157 | 2000 | 1.73 | ✓ |
| fiqa [176] | 130 MB | 25 k | 217 MB | 2974 | 13286 | 4.47 | ✓ |
| quora [177] | 641 MB | 523 k | 1.5 GB | 15672 | 30000 | 1.91 | ✓ |
| nq [178] | 4.6 GB | 2.68 M | 8.3 GB | 8186 | 10235 | 1.25 | ✗ |
| hotpotqa [165] | 11 GB | 5.42 M | 15.4 GB | 15519 | 22098 | 1.42 | ✗ |
| fever [179] | 7.5 GB | 5.23 M | 18.5 GB | 5783 | 13922 | 2.41 | ✗ |

redundant computation. Figure 4.6 shows the process of cluster embedding cache hit (4).

To optimize cache utilization, EdgeRAG strategically avoids caching embeddings from smaller clusters with lower generation latencies. This approach balances cache hit rates and overall latency. Caching all embeddings can lead to low hit rates and high latency, while exclusively caching expensive embeddings can improve hit rates but increase overall latency. An optimal threshold prevents caching of low-cost embeddings, striking a balance between the two.

To achieve this, EdgeRAG evicts and prevents caching of cluster embeddings whose generation latency falls below a dynamically adjusted Minimum Latency Caching Threshold. Algorithm 3 details how this threshold is adjusted. Initially, the threshold is set to 0, effectively caching all cluster embeddings. Subsequently, EdgeRAG gradually increases the threshold while continuously monitoring cache hit rates and the moving average of retrieval latency. If a cache miss occurs, and the current retrieval latency is lower than the moving average, the threshold is further increased. Conversely, if a cache miss does not occur, the threshold is decreased. This adaptive mechanism ensures that the cache prioritizes embeddings from clusters with significant generation costs to improve overall performance gains.

**Algorithm 2** Cost-aware Least-Frequently Used Replacement Policy

```
Incoming cache access with cluster index i
cluster = Cache.search(i)
if exist(cluster) then
    cluster.counter + +
else
    // Get Weighted Least Frequently Used Cluster
    minCost = MAXVALUE, evictClusterIndex = −1
    for cluster in Cache do
        if cluster.genLatency × cluster.counter < maxCost then
            minCost = cluster.genLatency × cluster.counter
            evictClusterIndex = cluster.index
        end if
    end for
    cache.delete(evictClusterIndex)
    // Get data chunks and generate embeddings
    dataChunks = getDataChunks(i)
    embeddings = embed(dataChunks)
    cache.insert(index = i, embeddings)
end if
// Update Counters
for cluster in Cache do
    cluster.counter = cluster.counter × decayFactor
end for
```

**Algorithm 3** Minimum Latency Caching Threshold

```
MinimumLatencyCachingThreshold = 0 // Initialize
for each Query do
    if embedCacheMiss = True then
        if movAvgLatency < lastLatency then
            MinimumLatencyCachingThreshold + +
        end if
    else
        MinimumLatencyCachingThreshold − −
    end if
    movAvgLatency = (1 − α) × movAvgLatency + α × lastLatency
end for
```

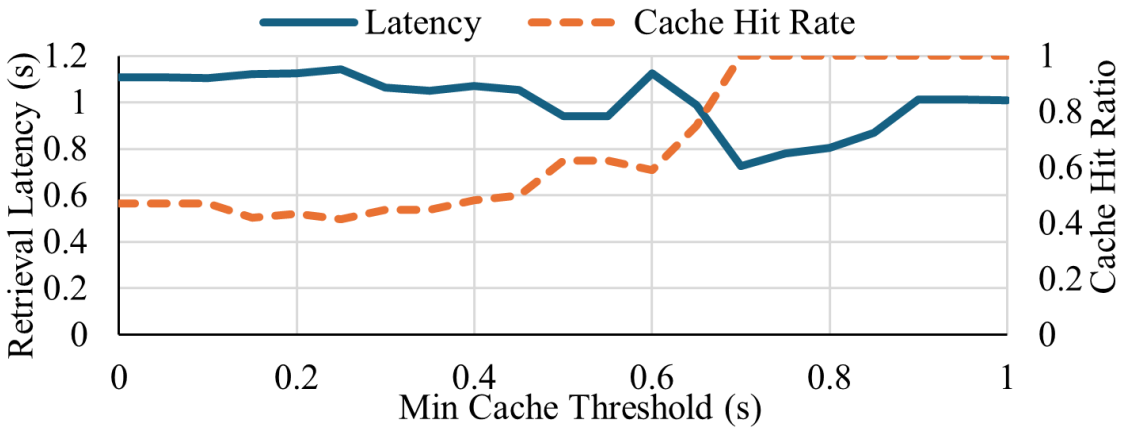



Figure 4.7: Retrieval Latency and Cache Hit rate with different Minimum Caching Threshold of fever dataset.

## 4.5 EdgeRAG System

### 4.5.1 Overview

EdgeRAG index is a two-level indexing system based on the traditional two-level Inverted File (IVF) Index. The first level, always residing in memory, stores cluster centroids and references to the second level index. The second level stores the references to the text chunks, and the embedding generation latency of all data chunks. However, instead of storing all text chunk embeddings, the embeddings are pruned and EdgeRAG generates them online during the retrieval process, only indexes of costly clusters are stored (see Section 4.4.1). To optimize performance and reduce latency, EdgeRAG employs a selective caching strategy for embeddings generated during the retrieval process (see Section 4.4.2). The system prioritizes caching expensive embeddings, where cache hits yield significant performance gains. Embeddings that can be regenerated quickly without compromising service-level objectives (SLOs) are avoided to make room for those more expensive embeddings.

### 4.5.2 EdgeRAG Indexing

EdgeRAG employs an indexing process similar to the Inverted File Index (IVF). Figure 4.8 illustrates the EdgeRAG indexing process. Initially, the text corpus is segmented into smaller data chunks (step ❶), and embeddings are generated for each chunk (step ❷). These embeddings are then clustered (step ❸), and the cluster centroids are stored in the first-level index (step ❹). Then the embeddings of the associated data chunks are assigned to their respective clusters (step ❺) and references to the corresponding data chunks are stored (step ❻). Unlike traditional IVF, where all data embeddings are retained, EdgeRAG calculates the computational cost of generating embeddings for each data chunk within a cluster. If this cost exceeds a predefined threshold (the Service Level Objective, or SLO), the embeddings of the whole data chunk are stored (step ❼). Otherwise, the embeddings are discarded to optimize storage.

### 4.5.3 EdgeRAG Retrieval

As mentioned in Section 4.4, EdgeRAG incorporates both heavy cluster embedding loading and embedding caching. Figure 4.9 shows the retrieval process EdgeRAG. EdgeRAG first identifies the centroid cluster most similar to the query embedding (Step ❶). It then checks if precomputed embeddings exist for this cluster (Step ❷). If so, EdgeRAG searches for stored embeddings (Step ❸) and these embeddings are loaded (Step ❺). If precomputed embeddings are unavailable,

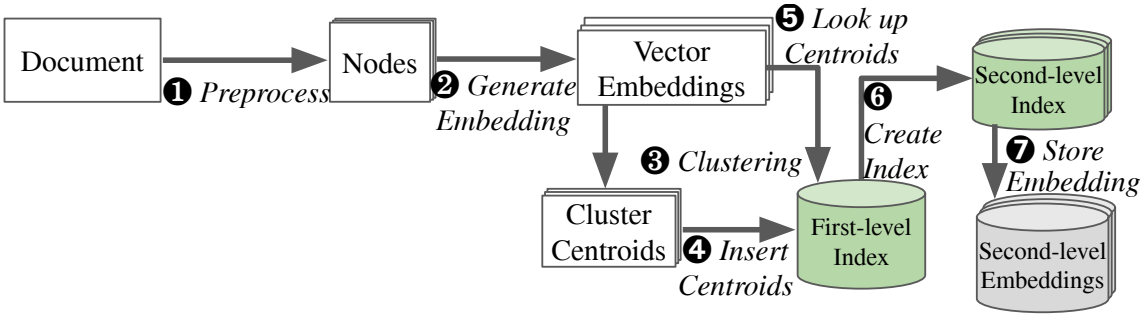



Figure 4.8: EdgeRAG Indexing Process

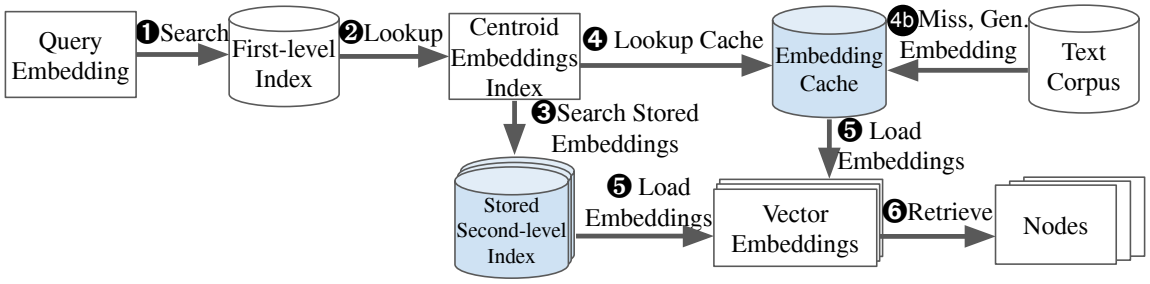



Figure 4.9: EdgeRAG Retrieval Process

EdgeRAG looks up the embedding cache (Step 4). If the cache *hits*, EdgeRAG loads the embedding from the cache and retrieves related data chunks (Step 5). If the cache *misses*, EdgeRAG retrieves all associated data chunks in the cluster, regenerates the embeddings, and updates the cache (Step 4b) before loading the embedding (Step 5). EdgeRAG then looks up for the closest matching embedding and retrieves associated data chunks (Step 6).

### 4.5.4 EdgeRAG Insertion and Removal

The EdgeRAG insertion process mirrors the initial indexing process. However, instead of clustering newly added embeddings, EdgeRAG identifies the existing cluster with the nearest centroid embedding. The index associated with this cluster is then updated. If the computational cost of the updated cluster's embeddings exceeds the SLO, EdgeRAG regenerates and stores its embeddings. In extreme cases where a cluster becomes excessively large, it is split into smaller clusters, and the newly created cluster is added to the first-level index. To remove data chunks, EdgeRAG first locates the corresponding cluster, then removes the associated embeddings, and the cluster index is subsequently updated. If the computational latency of generating the embedding of the cluster falls below the Service Level Objective (SLO), the entire cluster's embedding can be eliminated or merged with a neighboring cluster. This removal process can be performed asynchronously, as leaving the cluster

Table 4.3: Evaluation Platform.

| **Hardware System** | |
|---|---|
| CPU | Cortex A78AE, 1.2 GHz, 6-core |
| GPU | Ampere, 1024 CUDA cores, 32 Tensor cores, 625 MHz |
| Power Limit | 15 W |
| DRAM | 8 GB LPDDR5-4250 |
| Storage | 512 GB SD Card, UHS-I |
| **Software Configuration** | |
| RAG Framework | LlamaIndex v0.11.18 |
| Vector Store Engine | FAISS 1.7.4 |
| LLM Engine | NanoLLM 24.6 |
| Embedding Model | gte-base-en-v1.5 (Dim=768) |
| Generation Model | Sheared-LLaMA-2.7B |
| System Software | Jetpack 6.0 |

Table 4.4: Evaluated Index Configurations.

| Index configuration | Embeddings location | |
|---|---|---|
| | Level 1 | Level 2 |
| Flat | Memory | - |
| IVF | Memory | Memory |
| IVF+Embed. Gen. | Memory | - |
| IVF+Embed. Gen.+Load | Memory | Storage (Partial) |
| EdgeRAG (this work) | Memory | Storage + Memory |

small has no immediate adverse impact on retrieval latency.

## 4.6 Evaluation

### 4.6.1 System Setup

We evaluate EdgeRAG on an Nvidia Jetson Orin Nano evaluation kit. We leverage Llamaindex [180] as our RAG framework and FAISS [181] as our vector store. NanoLLM [182] powers our LLM engine, utilizing Sheared_LLaMA-2.7B [183] for generation. We employ gte-base-en-v1.5 [184] as our embedding model. Table 4.3 provides an overview of our evaluation setup and associated software environment.

### 4.6.2 Methodology

We evaluate EdgeRAG against five distinct configurations: a linear-search-based flat index (*Flat*), a two-level IVF index with precomputed cluster embeddings, a two-level index with online cluster embedding generation, a two-level index with online cluster embedding generation and large cluster loading from storage, and a two-level index with online cluster embedding generation, large cluster

loading from disk, and caching (*EdgeRAG*). Table 4.4 shows different evaluated Index configurations. For configurations utilizing the two-level IVF index, the embedding clustering process, performed using FAISS K-means with 20 iterations, is pre-computed and shared across all four configurations.

To address the known precision and recall trade-offs inherent to IVF-based indexing methods [174], we optimize the retrieval hyperparameters, specifically the number of cluster probes and retrieved data chunks. This optimization is aimed at normalizing the recall metric to match that of the flat index baseline [185]. We evaluate EdgeRAG on six datasets from the BEIR benchmark suite [171]. Table 4.2 provides a breakdown of the corpus size, number of data records, and total embedding size for each dataset. Note that three of the workloads: nq, hotpotqa, and fever have embedding footprints larger than the 8 GB memory capacity of our platform – impossible for the whole embedding database to fit into the memory. We set the retrieval SLO similar to those in LLM serving systems [186, 187]: 1 second for smaller *scidocs*, *fiqa* and *quora* datasets and 1.5 seconds for larger *nq*, *hotpotqa* and *fever* datasets.

### 4.6.3 Results

#### Retrieval Quality Evaluation

First, we evaluate the retrieval quality using the BEIR benchmark. We focus on Flat and two-level IVF indexes, as EdgeRAG, which optimizes the IVF index, produces identical retrieval results to the two-level IVF index. Figure 4.10 presents the precision and recall of different workloads. The results demonstrate a trade-off between accuracy and recall in two-level indexing schemes. While increasing the number of retrieved data chunks improves recall by including more relevant data, it also introduces more irrelevant data chunks, leading to a decrease in precision.

#### Generation Quality Evaluation

As mentioned in the previous section, IVF-based EdgeRAG exploits a precision-latency trade-off, prioritizing recall over precision. To investigate the potential benefits of prioritizing recall, we compare IVF-based EdgeRAG to a Flat index baseline. We employ a GPT-4o LLM [172] as an evaluator to assess generation quality across various datasets. As shown in Figure 4.11, while two-level IVF indexing schemes may exhibit lower precision, they can still achieve high-quality generation. This suggests that the generation model is capable of filtering out irrelevant information and leveraging only the most pertinent details for output generation.

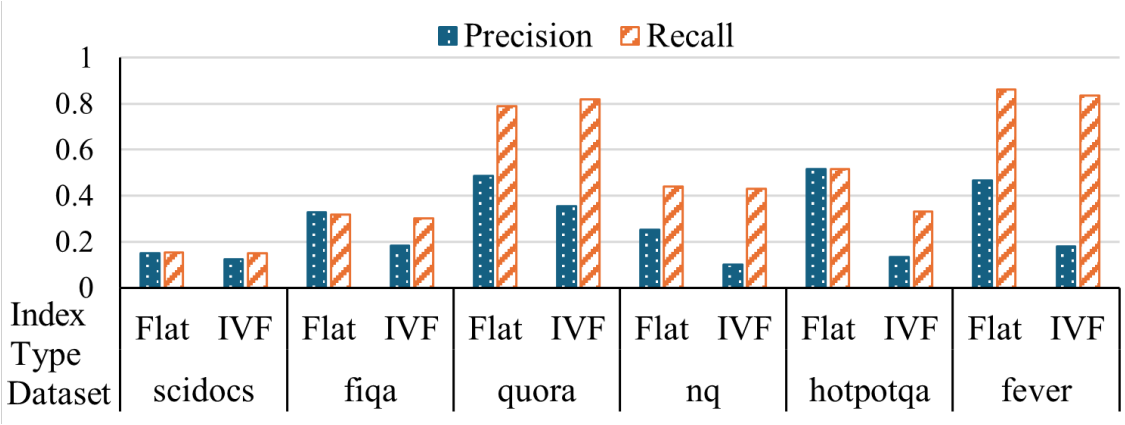



Figure 4.10: BEIR Evaluation Scores

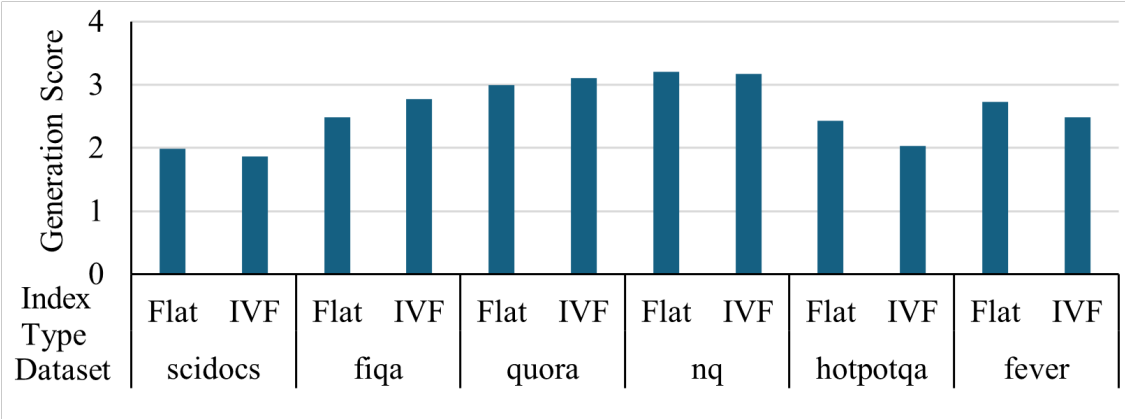



Figure 4.11: LLM Generation Evaluation Scores.

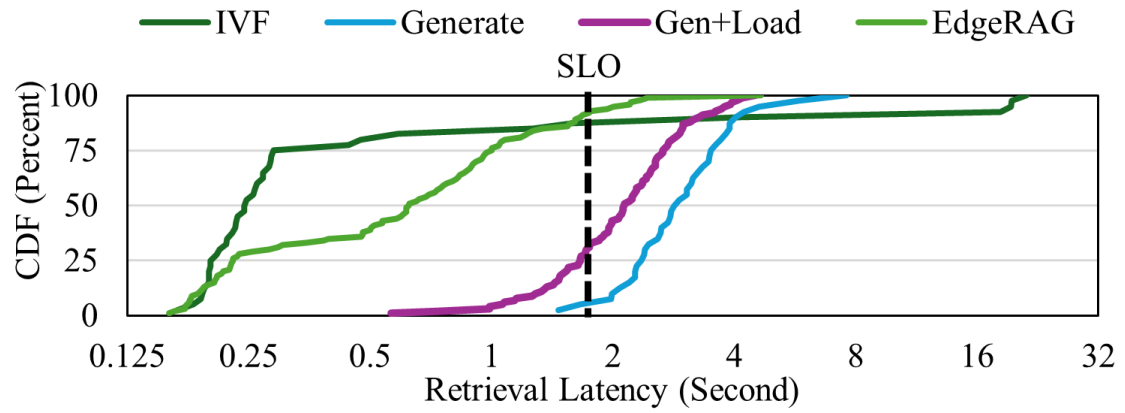



Figure 4.12: Retrieval Latency distribution with different optimizations.

### Retrieval Tail Latency Analysis

In this section, we evaluate the tail latency of different optimizations in EdgeRAG. Figure 4.12 illustrates the latency distribution of these optimizations for the nq dataset. The baseline IVF index,

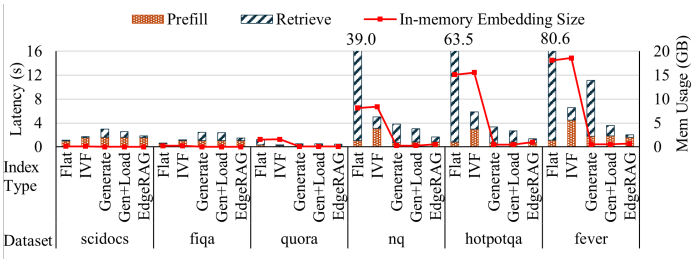



Figure 4.13: Retrieval and First Token Latency.

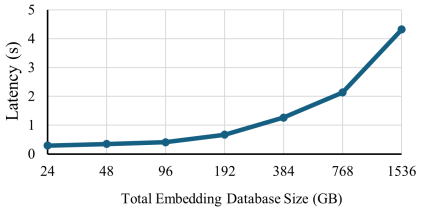



Figure 4.14: Retrieval Latency with different total embedding database sizes.

while achieving low latency for most queries, exhibits an extremely long tail latency, with the 95th percentile exceeding the median by over 64×. This behavior stems from memory thrashing when accessing paged-out cluster embeddings.

The IVF+Embed. Gen. optimization significantly improves latency by eliminating memory thrashing through embedding pruning. This results in a more than 4.22× reduction in 95th percentile latency compared to the IVF baseline. IVF+Embed. Gen. further optimizes with large cluster loading from storage yields another 1.218× tail latency reduction. Finally, caching eliminates redundant embedding computation, leading to significant overall latency reduction. While EdgeRAG prioritizes resource efficiency to meet retrieval SLOs, allocating more resources such as allocating more cache capacity could unlock even greater latency improvement.

### End-to-End Latency

In this section, we evaluate the end-to-end latency of a query. We use Time-to-First-Token (TTFT) as an evaluation metric. The TTFT consists of retrieval latency which includes vector similarity search, and prefill which is the time LLM takes to generate the first output token after it receives the prompt. We exclude Total Generation Time and Time-Per-Output-Token metrics as they are determined by the LLM's decoding rate, which is not optimized by EdgeRAG.

We then compare the latency of all five configurations (Table 4.4), as shown in Figure 4.13. For the two smallest datasets: scidocs and fiqa, where the entire embedding dataset can fit into memory, the online embedding generation scheme without caching exhibits higher retrieval latency compared to both Flat and baseline pre-generated Two-level indexes. This is because in-memory operations, even linear searches, can be executed very efficiently. For the Quora dataset, where the total embedding size nearly exceeds memory capacity, we observe that increasing the number of embeddings significantly degrades the performance of the Flat Index due to the overhead of linear search operations. For large datasets like nq, hotpotqa, and fever, where the total embedding size significantly exceeds the

memory capacity (8 GB in our platform), the baseline two-level IVF index suffers from increased retrieval latency and first token generation latency. This is due to the combined effects of memory thrashing during large index access and the eviction of the generation model from memory, leading to slower search and longer generation times.

In contrast, the EdgeRAG index, with and without heavy cluster loading and caching, does not thrash the memory and maintains low first token generation latency. However, for datasets with imbalanced, tail-heavy, and large clusters like Fever, the online cluster embedding generation process can introduce significant latency. Persisting heavy-tail cluster embeddings and employing caching can mitigate this issue by reducing tail latency and avoiding redundant embedding generation. This significantly improves overall retrieval latency 1.8 × on average while caching only utilizes an additional 7 % of system memory on top of a two-level online embedding generation scheme.

**Maximum Embedding Database Support**

In this experiment, we want to determine the maximum embedding database size that EdgeRAG can handle on our evaluation system (Table 4.3). We create a synthetic dataset by extracting text from the Quora dataset's text corpus and employ the same embedding model used in previous experiments. We choose EdgeRAG first-level index size to fully utilize our evaluation platform memory capacity and scale up the size of the dataset. We only evaluate *IVF+Embed. Gen* configuration (Table 4.4) in this experiment. Figure 4.14 shows that EdgeRAG supports embeddings database size up to 1.5 TB with only 8 GB of memory. By pruning all second-level embeddings, EdgeRAG achieves this remarkable scalability while consuming only 4.8 GB of memory for storing the embeddings. However, this optimization comes at a cost, resulting in a retrieval latency of 4.5 seconds. This experiment demonstrates EdgeRAG's ability to flexibly trade off latency for the capacity to handle massive embedding databases.

## 4.7 Discussion

**Limited Memory Capacity of Edge Systems** While some modern edge devices, such as upcoming mobile devices [188], boast substantial memory capacity, memory capacity still remains a major bottleneck for supporting large-scale datasets. EdgeRAG efficiently prunes the majority of second-level embeddings, enabling effective utilization of large memory capacities to support

larger datasets with more embedding footprints or more powerful LLMs for enhanced response quality.

**Integration with other RAG systems** In this work, we demonstrate EdgeRAG's effectiveness on a basic RAG system using text corpus data. While more advanced RAG systems may employ more sophisticated retrieval techniques or support diverse data types (multimodal), those techniques still rely on vector similarity search [189, 190]. As such, these systems can also benefit from EdgeRAG's optimizations.

**Exploiting Hardware Accelerator** Modern edge devices commonly integrate multiple hardware accelerators within their System on Chip [191, 192]. One of those is the Neural Processing Unit (NPU) which enables efficient processing of deep learning tasks. EdgeRAG can leverage these NPUs in several ways: 1. Enhance embedding generation throughput by harnessing the NPU's high throughput. 2. Offloading the embedding model's processing to the NPU—frees up the GPU or CPU for other tasks. This enables pipelining or parallelization of operations such as embedding generation, vector similarity search, and LLM prefill.

# Chapter 5

# Related Works

In this chapter, we present a survey of related works on RPC optimization, in-network compute and edge ML.

## 5.1 Datacenter Studies and RPC Optimizations

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.* [8], Alizadeh *et al.* [9], and Benson *et al.* [7]. 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.* [10] perform CPU profiling of Google's datacenters, and Gonzalez *et al.* [193] 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.* [194] 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.* [17] 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 [195], and Saokar *et al.* [195] found that the cloud-scale applications that use ServiceRouter show similar trends to those used at Google. Huye *et al.* [53] 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 [196]. 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 [15]. The RPC Chain is a new abstraction that can reduce network latency by chaining multiple RPC invocations [197]. 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 [198]. CRISP is a tool for analyzing RPC critical paths that was used at Uber to reduce tail latency [16]. 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 [38]. 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 [44], Raghavan *et al.* [45] and Karandikar *et al.* [72] introduce accelerators for marshalling/demarshalling

RPCs. Chiosa *et al.* [70] demonstrates an accelerator for offloading compression and encryption in SAP HANA database. Tonic [43] and AccelTCP [199] are hardware accelerators for the TCP transport protocol. Dagger offloads the entire RPC stack to an FPGA-based NIC [39]. NeBuLa is a CPU architecture optimized for accelerating microsecond-scale RPCs [40]. nanoPU creates faster paths from the network to CPU [51]. 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 $\mu$s-scale services. Although we found that most services at Google at not $\mu$s-scale, this does not mean that it is not possible to use $\mu$s-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 [200] and ServiceWeaver [201].

Shenango is a centralized software RPC load balancer, and RingLeader [46] and Turbo [202] are hardware accelerators for RPC load balancing. Shinjuku [42] and Caladan [59] 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.

**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 [198]. CRISP is a tool for analyzing RPC critical paths that was used at Uber to reduce tail latency [16]. eRPC is a system that onloads transport protocol processing to reduce latency [38].

**Accelerators for Transport and RPC stacks:** There have been many accelerators for transport and RPC stacks. For example, Zerializer [44], Raghavan *et al.* [45] and Karandikar *et al.* [72] introduce accelerators for marshalling/demarshalling RPCs. Chiosa *et al.* [70] demonstrates an accelerator for offloading compression and encryption in SAP HANA database. Tonic [43] and AccelTCP [199] are hardware accelerators for the TCP transport protocol. Dagger offloads the entire RPC stack to an FPGA-based NIC [39]. NeBuLa is a CPU architecture optimized for accelerating microsecond-scale RPCs [40]. nanoPU creates faster paths from the network to CPU to accelerate $\mu$s-scale services. [51]. Shenango is a centralized software RPC load balancer, and RingLeader [46] and Turbo [202] are hardware accelerators for RPC load balancing. Shinjuku [42] and Caladan [59] are CPU schedulers

that aim to isolate short- and long-running applications by reducing queuing delay for prioritized applications.

## 5.2 Optimization in Storage Systems and In-network Compute

Today, most of the computation takes place in these hyper-scale cloud data centers—performing more than 89% of the computation world-wide in 2018 [76].

These data centers host workloads ranging from time-critical, interactive jobs (e.g., online data-intensive (OLDI) workloads [14], RAMCloud [77, 78], and financial analysis [79]) to long-running, batch jobs (e.g., MapReduce [80] and machine-learning training [81]) with large memory footprints. In most of these workloads, data is typically managed and maintained in a persistent way across multiple servers, with clients accessing and updating this data remotely over a network of interconnected switches, using remote procedure calls (RPCs). During each invocation of an RPC, the request is processed by the client's IO stack, the network of intermediate switches, the server's IO stack as well as the request handler on the server. Thus, the latency of an RPC is significantly affected by the processing time of each of these stages. As the computation performed by modern workloads is dominated by these RPCs, i.e., read and update requests, the time it takes to access remote data is of major consideration when deploying workloads on modern data centers [82–86].

These RPCs can be either *synchronous* or *asynchronous*. Though, asynchronous RPCs can enable clients to continue execution while updates are being processed at the remote server; yet, building such applications is quite challenging, especially "at scale, when a typical end-to-end application can span multiple small closely interacting systems" [83]. In contrast, applications using synchronous RPCs are easy to write, tune, and debug—Google is known to strongly prefer a synchronous programming model [87, 88]. Therefore, in this work, our aim is to improve the performance (specifically the tail latency) for synchronous RPCs by minimizing the access time to remote persistent data.

Recently, as programmable network devices become available [89], a trend is to offload application logic to those devices. This way, a large fraction of the procedure, including server's network stack and processing time, is no longer handled by the server but accelerated by those network devices. This newer computation scheme is known as in-network compute, spanning a wide range of applications,

such as query processing [90, 91], key-value stores [11, 92–94], data aggregation [95–97], and even computational-intensive machine-learning tasks [98–100].

## 5.3 Persistent memory systems

The integration of PM improves the performance of managing persistent data over conventional storage devices, such as SSD and HDD. Various software systems take advantage of PM for better storage performance. PM-optimized file systems allow existing programs to manage data efficiently on PM [131, 134, 203]. Distributed, PM-based file systems allow clients to access PM remotely while benefiting from PM's high performance [135, 204, 205]. There are also PM-optimized databases and key-value store applications that accepts requests from clients [109–111, 206–208]. Applications can also choose to manage data on PM without software indirections, by maintaining their own PM data structures [209–215]. However, even with a faster storage backend, the client still needs to go through the network and wait for the entire RTT to update persistent data on the servers. In comparison, the integration of PMNet can improve the performance by moving both network stack and processing time off the critical path.

Performance of the PM system is one of the major aspects, and correctness is another, as ensuring a consistent recovery in PM systems is hard and error-prone. Recent works have provided tools that ensure PM-based applications recover to a consistent state in event of a failure [216–221]. These testing methods can be adapted to in-network data persistence systems, to validate not only the ordering in one application but also the persist ordering among clients and servers. Further, verification tools for programmable data plane [222, 223] can work in cooperation with PM testing tools and guarantee end-to-end correctness of a system with in-network data persistence. We leave this direction as a future work.

## 5.4 In-network compute

In-network compute reduces latency for a variety of tasks by moving computation off the server and into the network. Prior works proposed programmable switches for network functions such as load balancing [125, 126] and packet scheduling [129, 130]. In addition, programmable switches can also offload part of application's logic into the network such as data aggregation [95–97], and machine learning [96, 99, 100]. In-network compute can also accelerate storage workloads through caching [11, 94, 224]. However, the persistence domain in these workloads is still limited to the server.

A recent work NetChain [225] utilizes the storage capability of programmable switches; however, it only stores the coordination information rather than persisting data in the network. To maintain data persistence in the network, this work introduces PMNet that places PM on network devices, such as NICs and switches. We expect future works to further integrate other acceleration logic into PMNet to accelerate a wider range of applications.

## 5.5 Inference on edge devices

To handle limited resources on edge devices, several techniques have been proposed for on-device inference. Model compression simplifies models by either pruning parameters [226] or unnecessary connections in the model [227] or changing the structure of the model [228], leading to lower processing power and memory demands. Quantization reduces the precision of the model's values, primarily saving memory by using smaller data types [229]. Some specialized hardware even benefits from faster processing with these smaller data types [230]. Dynamic model execution adapts to the input. Early Exiting [231, 232] stops processing in Transformer models [233] once confidence reaches a threshold. OpenELM [234] adjusts the number of attention heads within a Transformer layer. As a last resort, highly demanding tasks can be partially offloaded to the cloud [235] to exploit the computing power of the cloud, albeit with network latency drawbacks. Finally, Retrieval Augmented Generation (RAG) [4] leverages external knowledge to assist LLM inference without constant retraining, but requires balancing the cost of generating and storage of a local index (embedding database) with search accuracy and response generation latency.

# Chapter 6

# Conclusions

Modern distribution applications, such as social networks, data analytics, and retrieval search, rely on both edge devices for low-latency processing and cloud systems for scalable performance. Optimizing these applications requires a comprehensive understanding of both edge and cloud components to effectively identify bottlenecks and direct optimization efforts. The first key challenge is that the performance characteristics of distributed cloud services are not yet well-understood. Second, storage services which underlie most services dominates datacenter RPC and are critical to cloud service performance. Techniques like caching and in-network computation can enhance performance for read requests. However, optimizing update requests remains challenging due to the risk of potential data loss. Third, edge applications, such as personal assistants that access locally stored data, require substantial data processing to retrieve information and generate responses. Although offloading this processing to the cloud is feasible, it introduces risks related to personal data leakage. Conversely, performing this computation locally on edge devices presents significant challenges due to resource constraints, particularly limited memory capacity.

In order to overcome these challenges, we hypothesize that comprehensive understanding of end-to-end performance in modern distributed systems could enable targeted optimization at the critical bottlenecks.

In order to understand the characteristics of cloud based services, we perform the first analysis of RPCs from geo-distributed applications running across a fleet of datacenters. We evaluate RPC invocations within Google's internal production system. Our study analyzes over 700 billion RPC

traces from over 10000 different RPC functions running on over 100 production clusters. This data allows us to unveil RPC's key properties such as latency, volume and call structures. 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.

For storage systems, we propose in-network data persistence that exposes the persistence domain to the network to persist update requests with sub-RTT latency. We design a PM-integrated programmable network device, PMNet, that logs in-flight update requests, and moves the server network stack and processing time off the critical path. We implement PMNet in an FPGA-based programmable switch and NIC and evaluate them in a real system with a variety of workloads. Compared to the existing system, PMNet can improve the throughput of update requests by 4.31× on average, and the 99th-percentile tail latency by 3.23×. PMNet could also synergistically integrate with read-caching mechanisms to accelerate read requests and provide in-network replication of data to protect again possible hardware and network failures.

For edge systems, we propose EdgeRAG, a novel RAG system designed to address the memory limitations of edge platforms. EdgeRAG optimizes the two-level IVF index by pruning unnecessary second-level embeddings, selectively storing or regenerating them during execution, and caching generated embeddings to minimize redundant computations. This approach enables efficient RAG applications on datasets that exceed available memory, while preserving low retrieval latency and without compromising generation quality. Our evaluation results show that EdgeRAG improves retrieval latency by 1.22× on average and by a substantial 3.69× for large datasets that cannot fit in the memory. With relaxed latency SLO, EdgeRAG can support up to 1.5 TB of embedding database on system with 8GB memory.

The findings in this work also inspire new research directions.

- **RPC improvement:** Our measurements of the largest-scale cloud applications could set the direction of RPC systems development in the future. Particularly, our RPC latency

measurement shows that most RPC functions operate in millisecond timescale, the RPC optimization should balance between latency and RPC resource efficiency. Queuing and scheduling latency are significant for some types of applications, improved scheduling and load balancing have significant potential to improve these applications.

- **In-network storage acceleration:** Our work, PMNet, demonstrates a systematic approach enabling network devices to durably store data despite potential failures. This durability ensures that in-network computation can handle not only stateless requests but also longer, stateful computing tasks that might otherwise be disrupted by failures.

- **ML inference on Edge Devices:** EdgeRAG overcomes the memory limitation of traditional RAG techniques and enables the edge-local integration of large databases with large language models previously only available in the cloud.

# Bibliography


[1] Salvatore Sanfilippo. antirez/retwis.

[2] Transaction Processing Performance Council (TPC). TPC-C.

[3] Intel. Intel Optane DC persistent memory.

[4] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[12] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.

[13] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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/`.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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 *Pro-*

*ceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

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

[50] 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.

[51] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A nanosecond network stack for data-

centers. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2021.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Markets and Markets. Data center power market by solution (power distribution & measurement, power backup, cabling infrastructure), service (system integration, training & consulting, support & maintenance), end-user type, vertical, and region - global forecast to 2021, 2017.

[76] Yevgeniy Sverdlik. Study: Data centers responsible for 1 percent of all electricity consumed worldwide, 2020. `https://www.datacenterknowledge.com/energy/study-data-centers-responsible-1-percent-all-electricity-consumed-worldwide`.

[77] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 2015.

[78] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[79] Ramana Rao Kompella, Kirill Levchenko, Alex C. Snoeren, and George Varghese. Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2009.

[80] 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.

[81] Martín Abadi et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[82] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[83] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 2017.

[84] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2015.

[85] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2019.

[86] Mohammad Alian and Nam Sung Kim. NetDIMM: Low-latency near-memory network interface architecture. In *MICRO*, 2019.

[87] Jeff Shute et al. F1: A distributed SQL database that scales. *VLDB*, 2013.

[88] James C Corbett et al. Spanner: Google's globally distributed database. *TOCS*, 2013.

[89] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2013.

[90] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2018.

[91] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. The case for network accelerated query processing. In *Proceedings of the 9th Conference on Innovative Data Systems Research (CIDR)*, 2019.

[92] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2017.

[93] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. LaKe: An energy efficient, low latency, accelerated key-value store. *CoRR*, abs/1805.11344, 2018.

[94] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[95] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[96] Fan Yang, Zhan Wang, Xiaoxiao Ma, Guojun Yuan, and Xuejun An. SwitchAgg: A further step towards in-network computation. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[97] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*. Association for Computing Machinery, 2017.

[98] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2019.

[99] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale DNN

processor for real-time AI. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2018.

[100] Youjie Li et al. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *MICRO*, 2018.

[101] Jeff Barr. Now available – Amazon EC2 high memory instances with 6, 9, and 12 TB of memory, perfect for SAP HANA, 2018.

[102] Nan Boden. Available first on Google cloud: Intel Optane DC persistent memory.

[103] Arthur Sainio. NVDIMM - Changes are here so what's next?, 2016.

[104] Dell. Dell EMC PowerEdge R740xd Intel Optane, 2019.

[105] Lenovo. Analyzing the performance of Intel Optane DC persistent memory in storage over app direct mode, 2019.

[106] Joseph Izraelevitz et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv*, 2019.

[107] Virendra J. Marathe et al. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *HotStorage*, 2017.

[108] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, 2017.

[109] Lenovo. Memcached-pmem, 2018.

[110] Intel. Redis, 2018.

[111] Fei Xia et al. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *ATC*, 2017.

[112] Xianzhang Chen et al. UDORN: A design framework of persistent in-memory key-value database for NVM. In *NVMSA*, 2017.

[113] Xingbo Wu et al. NVMcached: An NVM-based key-value cache. In *ApSys*, 2016.

[114] Akshitha Sriraman and Thomas F. Wenisch. µTune: Auto-tuned threading for OLDI microservices. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2018.

[115] Yu Gan et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[116] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[117] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2016.

[118] Alexandros Daglis et al. RPCValet: NI-drive tail-aware balancing of ms-scale RPCs. In *ASPLOS*, 2019.

[119] Alexandros Daglis et al. Manycore network interfaces for in-memory rack-scale computing. In *ISCA*, 2015.

[120] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F. Wenisch. Deconstructing the tail at scale effect across network protocols. *CoRR*, abs/1701.03100, 2017.

[121] Intel. Persistent memory programming.

[122] Microsoft. Asynchronous programming - c#.

[123] Barefoot Networks. Tofino.

[124] Xilinx. X2 series Ethernet adapters.

[125] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2017.

[126] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research (SOSR)*, 2016.

[127] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2017.

[128] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[129] J. Guo, J. Yao, and L. Bhuyan. An efficient packet scheduling algorithm in network processors. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM)*, 2005.

[130] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2016.

[131] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

[132] Jian Xu et al. Nova-Fortis: A fault-tolerant non-volatile main memory file system. In *SOSP*, 2017.

[133] Harendra Kumar et al. High-performance metadata integrity protection in the WAFL copy-on-write file system. In *FAST*, 2017.

[134] Subramanya R Dulloor et al. System software for persistent memory. In *EuroSys*, 2014.

[135] Youyou Lu et al. Octopus: An RDMA-enabled distributed persistent memory file system. In *ATC*, 2017.

[136] Haris Volos et al. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[137] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *SC*, 2011.

[138] Youngjin Kwon et al. Strata: A cross media file system. In *SOSP*, 2017.

[139] Mellanox. Messaging accelerator (VMA), 2020.

[140] Jeffrey Dean et al. Large scale distributed deep networks. In *NeurIPS*, 2012.

[141] Charles Reiss et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[142] Berk Atikoglu et al. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[143] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[144] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable ASICs. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*. Association for Computing Machinery, 2018.

[145] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2013.

[146] L. A. Barroso, U. Hölzle, P. Ranganathan, and M. Martonosi. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. 2018.

[147] Bettina Kemme and Gustavo Alonso. Database replication: A tale of research across communities. *Proceedings of the VLDB Endowment*, 2010.

[148] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2013.

[149] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery, 2014.

[150] Z. Wang et al. Characterizing and modeling non-volatile memory systems. In *MICRO*, 2020.

[151] NetFPGA. NetFPGA-SUME Virtex-7 FPGA development board.

[152] 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. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[153] Brian F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[154] Dulcardo Arteaga and Ming Zhao. Client-side flash caching for cloud systems. In *SYSTOR*, 2014.

[155] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. Request-oriented durable write caching for application performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2015.

[156] Shimin Chen et al. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD*, 2011.

[157] IBM. How to reset/reboot server iMM interface.

[158] Andy Rudoff. Persistent memory programming without all that cache flushing.

[159] Emre Kültürsay et al. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[160] Cong Xu et al. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, 2015.

[161] Tom Talpey. SNIA nonvolatile memory programming TWG – Remote persistent memory, 2019.

[162] Chris Mellor. Toshiba nudges direct-to-ethernet SSD towards the light, 2019.

[163] Google. Google assistant, 2024.

[164] Microsoft. Microsoft Copilot: Your AI companion.

[165] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. FEVER: a large-scale dataset for fact extraction and VERification. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 809–819, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[166] Gary Sims. How much ram does your android phone really need in 2024?, Oct 2024.

[167] Kyle Wiens. More modular than ever before: iPhone 16 Pro and Pro Max teardown, Sep 2024.

[168] Samsung. What are the S24 external memory limits and memory sizes of different S24 variants?, Jan 2024.

[169] NVIDIA. NVIDIA Jetson Orin.

[170] NVIDIA. NVIDIA L40.

[171] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In

*Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[172] OpenAI. Hello gpt-4o.

[173] Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. Ares: An automated evaluation framework for retrieval-augmented generation systems. *arXiv preprint arXiv:2311.09476*, 2023.

[174] Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1470–1477 vol.2, 2003.

[175] Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S. Weld. Specter: Document-level representation learning using citation-informed transformers. In *ACL*, 2020.

[176] Macedo Maia, Siegfried Handschuh, André Freitas, Brian Davis, Ross McDermott, Manel Zarrouk, and Alexandra Balahur. Www'18 open challenge: Financial opinion mining and question answering. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 1941–1942, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[177] Shankar Iyer, Nikhil Dandekar, and Kornel Csernai. First quora dataset release: Question pairs.

[178] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[179] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[180] Jerry Liu. LlamaIndex, 11 2022.

[181] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[182] Dustin Franklin. Nanollm: Optimized local inference for llms.

[183] Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. Sheared llama: Accelerating language model pre-training via structured pruning. *arXiv preprint arXiv:2310.06694*, 2023.

[184] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning, 2023.

[185] Qi Chen, Xiubo Geng, Corby Rosset, Carolyn Buractaon, Jingwen Lu, Tao Shen, Kun Zhou, Chenyan Xiong, Yeyun Gong, Paul Bennett, et al. Ms marco web search: a large-scale information-rich web dataset with millions of real click labels. In *Companion Proceedings of the ACM on Web Conference 2024*, pages 292–301, 2024.

[186] Andreas Kosmas Kakolyris, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Slo-aware gpu dvfs for energy-efficient llm inference serving. *IEEE Computer Architecture Letters*, 2024.

[187] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[188]

[189] Wenhu Chen, Hexiang Hu, Xi Chen, Pat Verga, and William W Cohen. Murag: Multimodal retrieval-augmented generator for open question answering over images and text. *arXiv preprint arXiv:2210.02928*, 2022.

[190] Weizhe Lin and Bill Byrne. Retrieval augmented visual question answering with outside knowledge. *arXiv preprint arXiv:2210.03809*, 2022.

[191] Monika Gupta. Google tensor is a milestone for machine learning. *Google*, October 2021.

[192]

[193] 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.

[194] 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.

[195] 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.

[196] 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.

[197] 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.

[198] 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.

[199] 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.

[200] 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.

[201] Google. Service weaver. `https://serviceweaver.dev/`.

[202] 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.

[203] Jeremy Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[204] Yiying Zhang et al. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.

[205] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *SoCC*, 2017.

[206] Intel. Pmem-rocksdb.

[207] Intel. PMSE - Persistent memory storage engine for MongoDB, 2018.

[208] Katelin A. Bailey et al. Exploring storage class memory with key value stores. In *INFLOW*, 2013.

[209] Qingda Hu et al. Log-structured non-volatile main memory. In *ATC*, 2017.

[210] Diego Ongaro et al. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[211] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. In *VLDB*, 2015.

[212] Jun Yang et al. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, 2015.

[213] Nachshon Cohen et al. Object-oriented recovery for non-volatile memory. *OOPSLA*, 2018.

[214] Joy Arulraj et al. Bztree: A high-performance latch-free range index for non-volatile memory. *VLDB*, 2018.

[215] Shivaram Venkataraman et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[216] Sihang Liu et al. PMTest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.

[217] Sihang Liu et al. Cross-failure bug detection in persistent memory programs. In *ASPLOS*, 2020.

[218] Sihang Liu et al. PMFuzz: Test case generation for persistent memory programs. In *ASPLOS*, 2021.

[219] Bang Di et al. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *ASPLOS*, 2021.

[220] Ian Neal et al. AGAMOTTO: How persistent is your persistent memory application? In *OSDI*, 2020.

[221] Hamed Gorjiara et al. Jaaru: Efficiently model checking persistent memory programs. In *ASPLOS*, 2021.

[222] Jed others Liu. P4v: Practical verification for programmable data planes. In *SIGCOMM*, 2018.

[223] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *SOSR*, 2018.

[224] Mengxing Liu et al. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.

[225] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[226] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[227] Ivan Lazarevich, Alexander Kozlov, and Nikita Malinin. Post-training deep neural network pruning via layer-wise calibration. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 798–805, 2021.

[228] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16091–16101, 2023.

[229] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[230] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings.(2023). *arXiv preprint arXiv:2304.01433*, 2023.

[231] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[232] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

[233] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[234] Sachin Mehta, Mohammad Hossein Sekhavat, Qingqing Cao, Maxwell Horton, Yanzi Jin, Chenfan Sun, Iman Mirzadeh, Mahyar Najibi, Dmitry Belenko, Peter Zatloukal, et al.

Openelm: An efficient language model family with open-source training and inference framework. *arXiv preprint arXiv:2404.14619*, 2024.

[235] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.