Optimizing Distributed Workloads with Infrastructure-managed Communication and Deployment

by

Yongji Wu

Department of Computer Science
Duke University

Defense Date: November 18, 2024

Approved:

Danyang Zhuo, Supervisor

Matthew Lentz, Supervisor

Jeffrey Chase

Bhuwan Dhingra

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2025

ABSTRACT

Optimizing Distributed Workloads with Infrastructure-managed Communication and Deployment

by

Yongji Wu

Department of Computer Science
Duke University

Defense Date: November 18, 2024

Approved:

Danyang Zhuo, Supervisor

Matthew Lentz, Supervisor

Jeffrey Chase

Bhuwan Dhingra

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2025

Abstract

As the scale and complexity of distributed workloads grows, performance is no longer the sole objective sought by application developers and infrastructure operators, as they increasingly demand cost efficiency and manageability. Existing system infrastructure struggles to meet these goals. On the lower-level datacenter network stacks, existing solutions rely on a library-based approach where tenants implement and control communication of their workloads. Without insights of the infrastructure and other tenants, they achieve sub-optimal performance while offering limited manageability in an inefficient way. On the higher-level application deployment side, the space of deployment configuration has grown intractably large for users to manually tune, especially with new workloads like machine learning inference workflows and new infrastructure options like spot instances.

In this dissertation, I argue that decoupling the implementation of communication primitives and the control of deployment strategies from distributed applications can improve their performance, cost efficiency, and manageability. On the lower-level communication side, we can implement common primitives via managed system services provided by the infrastructure operators, enabling new performance optimization opportunities and better manageability with negligible overheads. On the higher-level workload deployment side, we can build systems that manage and optimize deployment strategies for new workloads on new types of infrastructure, improving cost efficiency without sacrificing performance.

The contributions of this dissertation are the design, implementation and evaluation of the following systems. (1) To improve the performance of remote procedure calls (RPCs) and enhance manageability, we present mRPC, a system service that decouples RPC marshalling and policy enforcement from applications, speeding up microservice applications by up to 2.5x compared to existing solutions for enforcing polices. (2) To improve the performance and manageability of collective communication, we introduce MCCS, a system service that exposes collective communication abstractions to applications while providing control and flexibility to cloud providers for their implementation, improving tenant collective performance by up to 2.4x compared with existing library based solutions. (3)

To improve the performance and cost efficiency when deploying machine learning inference workflows, we develop, JellyBean, a system service that optimizes and serves them over heterogeneous infrastructure, reducing total serving cost by up to 58%; (4) To improve the performance and cost efficiency for training mixture-of-experts (MoE) models, we build Lazarus, a system service that manages and optimizes training of MoE models on spot instances with resiliency and elasticity, enabling cost reductions while outperforming existing checkpoint-based systems by up to 3.4x.

Dedication

Dedicated to my parents for their love and support.

Contents

List of Tables

List of Figures

Acknowledgements

Looking back four years later since the start of my PhD, it has been an incredible journey, filled with both challenges and joy. This journey would not be possible without the collective support, guidance and encouragement from many remarkable individuals.

First and foremost, I would like to express my deepest gratitude to my advisors, Danyang Zhuo and Matthew Lentz. They have always been there to offer me invaluable insights, guidance and unwavering support, both in research and in life. From a novice who know little about system research, they directed me towards important problems, taught me how to transform raw ideas to solid implementations and papers, guided me to become an independent researcher. They have always instilled optimism in me whenever I encountered challenges and hardships. They are not only exceptional advisors, but also trusted friends to their students. I could not be any luckier for having them as my advisors.

I would also like to thank my committee members on my preliminary exam and defense: Jeffrey Chase, Bbhuwan Dhingra and Sam Wiseman, for their invaluable feedback.

I am very fortunate to be surrounded by many excellent research collaborators: Jingrong Chen, Ceyu Xu, Xinhao Kong, Yechen Xu, Shihan Lin, Yicheng Jin, Jianxin Qin, Xiaoyu Cao, Prof. Xiaowei Yang and Prof. Neil Gong at Duke; Xinyu Yang and Prof. Beidi Chen at CMU; Wenjie Qu, Tianyang Tao, Prof. Jiaheng Zhang and Prof. Yao Lu at NUS; Shuowei Jin, Xueshen Liu, Haizhong Zheng, Qingzhao Zhang, Prof. Morley Mao and Prof. Atul Prakash at UMich; Prof. Feng Qian at USC; Lequn Chen, Zihao Ye, Prof. Thomas Anderson, Prof. Luis Ceze and Prof. Arvind Krishnamurthy at UW; Wei Bai at NVIDIA; Zhaodong Wang and Ying Zhang at Meta; Zhuang Wang at AWS. Many of my projects would not be possible without their support and help.

I am also thankful to my undergraduate advisor, Defu Lian, who introduced me to the world of research. Defu helps me understand what is research and guides me towards my first research paper. The skills and insights I gained from him laid the foundation for my PhD.

I also want to extend my gratitude to all my other friends, for both research discussions

and non-research discussions, including Jiaao Ma, Zonghao Huang, Yiheng Xu, Hejie Cui, Chulin Xie, Shinan Liu, Shi Liu, Mingji Han, Jinwei Yao, Jinghuai Zhang, Wenxuan Wang, Bo Meng, Chi Zhang, Aobo Liu, Guozhen She, Lianke Qin, Mingtao Chen and Yuhui Li. I also especially thanks Shuowei, as he has always provided me with emotional support and a source of strength through all my highs and lows since undergraduate.

Lastly, my sincere thanks go to my parents for always supporting and believing in me. I am also grateful to the company of my girlfriend, Zifu.

1. Introduction

We have witnessed a significant growth in the scale of distributed workloads. For example, Uber's infrastructure is composed of 4500 microservices and they are deployed more than 100,000 times each week [278]. Uber's data platform hosts 1.5 exabytes of data using Hadoop Distributed File System and serves over 370,000 Spark applications daily [277]. Besides these traditional datacenter workloads, machine learning workloads have significantly increased in the amount of compute, especially with the rise of large language models. For instance, the 405B Llama 3 model is trained with a cluster of 16K GPUs [48] and requires a total training time of 30.84M GPU hours [172].

The complexity in deploying these workloads also dramatically increased. For instance, in the era of vision models, they are generally trained using pure data parallelism. However, today's LLMs are frequently beyond the memory capacity of a single GPU. They are generally trained with a combination of data, tensor and pipeline parallelism [187], where the space of parallelism configurations is tremendously increased. With the emergence of sky computing [296] and the advent of edge computing, distributed workloads may be deployed on heterogeneous infrastructure, across edge and multiple clouds.

Although *performance* has always been a key optimization objective when deploying these workloads, other factors like *cost efficiency* and *manageability* have recently gained increasing attention. With the growing scale and complexity of distributed workloads, the cost of running them has also surged significantly. It is estimated the training of the state-of-the-art GPT-4 model costs around $100 million [289], while it takes about $700,000 per day for OpenAI to serve ChatGPT [247]. With such exorbitant costs, both the infrastructure operators and application developers aim to improve not only the *performance* of distributed workloads, but also their *cost efficiency*. In addition, infrastructure operators have expressed a growing demand for a higher degree of *manageability*. For instance, there is a need to monitor and control the performance of certain Remote Procedure Calls (RPCs) [22] between microservices developed by different teams [177], and infrastructure operators may impose certain rate limiting and access control policies. For compute and

communication intensive workloads like ML training, operation teams may impose certain quality of service (QoS) polices, enforcing different SLOs (e.g., deadlines) for different models being trained [75, 63, 149].

Still, existing system stacks struggle to meet these goals. There are two primary challenges to address. First, *there is information asymmetry between the applications and the infrastructure providers.* Second, *the deployment choices of the workloads have expanded exponentially*, as both the complexity of the workloads themselves and infrastructure grows.

Information asymmetry between applications and infrastructure providers. With the advent of server virtualization, multi-tenant datacenters and public cloud providers expose a black-box model for tenants to access to compute and network resources, with resource allocation (e.g., vCPU, GPU, memory and network bandwidth) at per virtual machine level. In existing datacenter system stacks, the network communication is handled by tenants themselves using a library-based approach, where common communication primitives like RPC and collective communication are implemented in libraries linked to tenants' applications. On the one hand, tenant applications often make sub-optimal choices based on fundamentally incomplete information. They are unaware of the information available to the infrastructure providers, such as the physical cluster topology and workloads from other tenants. On the other hand, infrastructure providers often have little to no observability and control to the characteristics and behaviors of the applications.

In terms of performance, many optimizations of a distributed workload's communication strategies requires the information of the physical topology. Such strategy choices would greatly impact the performance of the workloads, especially for communication-intensive workloads like ML training. In a single tenant scenario where the developers also control the infrastructure, they can manually optimize their workloads accordingly; although it requires strong system expertise. In a multi-tenant environment, however, such topology information are not available to the tenants. As a consequence, this may leads to sub-optimal communication paradigms, impairing the performance of the workloads. In

addition, the optimal communication strategy for a workload also depends on other workloads' communication patterns. Without a holistic view of all tenants when choosing the strategy, different workloads may compete on shared resources (i.e., network links) and lead to overall performance degradation [230, 54].

In terms of manageability, although existing solutions offer limited support for certain types of communication primitive used in distributed workloads, they have significant overheads. For instance, a sidecar based architecture is typically used to enforce polices for RPCs. With only a weak coupling to the application, a sidecar intercepts, processes and forwards RPC messages via the operating system's network stack. This approach leads to many redundant marhsalling and unmarshalling steps, resulting in an overhead of up to 2.8x in terms of tail latency and 0.56x in terms of goodput. For other communication primitives like collective communication, there completely lacks a solution for such policy enforcement.

In summary, with information asymmetry, existing tenant controlled library-based solutions for datacenter communication achieve *subpar performance* and offer *limited manageability in an inefficient way.*

Complex space of deployment choices of the workloads. In terms of workflows, for instance, the landscape of machine learning has evolved from a single small model to complex ML workflows composed of multiple ML operations [186] using inputs from different sources, and significantly larger LLMs. The infrastructure is also becoming more heterogeneous. On the one hand, with the advent of Internet of Things (IoT) devices like smart cameras and speakers, complex workflows can partially utilize their varying on-board computing capabilities. On the other hand, cloud now offers many different types of GPUs and even options like spot instances that trade-offs availability for cost savings. Considering all these complexities in the workloads and infrastructure leads to such a large search space, making it intractably difficult for developers to manually tune the deployment configuration. To enable application developers quickly deploy their workloads efficiently, we need system

infrastructure that automatically optimizes the deployment strategies. However, existing systems *fail to apply to new types of workloads and take advantage of new infrastructure options.* For instance, existing ML serving systems only target a single model or a set of models on homogenous infrastructure.

1.1 Dissertation Statement

Given these chanllenges, in this dissertation, I argue that *decoupling the implementation of communication primitives and the control of deployment strategies from distributed applications can improve their performance, cost efficiency, and manageability.*

1.2 Dissertation Contributions

This dissertation makes the following contributions:

- Decoupling communication for performance and manageability: we rearchitect RPC to decouple RPC marshalling and policy enforcement into a managed service, improving manageability with negligible overheads.
- Decoupling communication for performance and manageability: we rearchitect collective communication to decouple the implementation of various collective algorithms into a managed service, for cloud providers to control and optimize collective traffic, enabling novel performance optimizations and enhancing manageability.
- Decoupling application deployment for performance and cost efficiency: we design a system that automatically optimizes and serves ML inference workflows on heterogeneous infrastructures, reducing serving costs.
- Decoupling application deployment for performance and cost efficiency: we build a system that takes advantage of spot instances for cost-efficient training of mixture-of-experts (MoE) models.

Decoupling RPC implementation into a managed service. Remote Procedure Call (RPC) is a widely used abstraction for cloud computing. The programmer specifies type information for each remote procedure, and a compiler generates stub code linked into each

application to marshal and unmarshal arguments into message buffers. Increasingly, however, application and service operations teams need a high degree of visibility and control over the flow of RPCs between services, leading many installations to use sidecars or service mesh proxies for manageability and policy flexibility. These sidecars typically involve inspection and modification of RPC data that the stub compiler had just carefully assembled, adding needless overhead. Further, upgrading diverse application RPC stubs to use advanced hardware capabilities such as RDMA or DPDK is a long and involved process, and often incompatible with sidecar policy control.

We propose, implement, and evaluate a novel approach, where RPC marshalling and policy enforcement are done as a system service rather than as a library linked into each application. Applications specify type information to the RPC system as before, while the RPC service executes policy engines and arbitrates resource use, and then marshals data customized to the underlying network hardware capabilities. Our system, mRPC, also supports live upgrades so that both policy and marshalling code can be updated transparently to application code. Compared with using a sidecar, mRPC speeds up a standard microservice benchmark, DeathStarBench, by up to 2.5× while having a higher level of policy flexibility and availability.

Decoupling collective communication implementation into a managed service. Performance of collective communication is critical for distributed systems. Using libraries to implement collective communication algorithms is not a good fit for a multi-tenant cloud environment because the tenant is not aware of the underlying physical network configuration or how other tenants use the shared cloud network—this lack of information prevents the library from selecting an optimal algorithm. We explore a new approach for collective communication that more tightly integrates the implementation with the cloud network instead of the applications. We introduce MCCS, or Managed Collective Communication as a Service, which exposes traditional collective communication abstractions to applications while providing control and flexibility to the cloud provider for their implementations.

Realizing MCCS involves overcoming several key challenges to integrate collective communication as part of the cloud network, including memory management of tenant GPU buffers, synchronizing changes to collective communication strategies, and supporting policies that involve cross-layer traffic optimization. Our evaluations show that MCCS improves tenant collective communication performance by up to 2.4× compared to one of the state-of-the-art collective communication libraries (NCCL), while adding more management features including dynamic algorithm adjustment, quality of service, and network-aware traffic engineering.

Serving ML workflows on heterogeneous infrastructures. With the advent of ubiquitous deployment of smart devices and the Internet of Things, data sources for machine learning inference have increasingly moved to the edge of the network. Existing machine learning inference platforms typically assume a homogeneous infrastructure and do not take into account the more complex and tiered computing infrastructure that includes edge devices, local hubs, edge datacenters, and cloud datacenters. On the other hand, recent AutoML efforts have provided viable solutions for model compression, pruning and quantization for heterogeneous environments; for a machine learning model, now we may easily find or even generate a series of model variants with different tradeoffs between accuracy and efficiency.

We design and implement JellyBean, a system for serving and optimizing machine learning inference workflows on heterogeneous infrastructures. Given service-level objectives (e.g., throughput, accuracy), JellyBean picks the most cost-efficient models that meet the accuracy target and decides how to deploy them across different tiers of infrastructures. Evaluations show that JellyBean reduces the total serving cost of visual question answering by up to 58% and vehicle tracking from the NVIDIA AI City Challenge by up to 36%, compared with state-of-the-art model selection and worker assignment solutions. JellyBean also outperforms prior ML serving systems (e.g., Spark on the cloud) up to 5x in serving costs.

System for MoE models training on spot instances. Sparsely-activated Mixture-of-Experts (MoE) architecture has increasingly been adopted to further scale large language models (LLMs) due to its sub-linear scaling for computation costs. However, frequent failures still pose significant challenges as training scales. The cost of even a single failure is significant, as all GPUs need to wait idle until the failure is resolved, potentially losing considerable training progress as training has to restart from checkpoints. Existing solutions for efficient fault-tolerant training either lack elasticity or rely on building resiliency into pipeline parallelism, which cannot be applied to MoE models due to the expert parallelism strategy adopted by the MoE architecture.

We present Lazarus, a system for resilient and elastic training of MoE models. Lazarus adaptively allocates expert replicas to address the inherent imbalance in expert workload and speeds-up training, while a provably optimal expert placement algorithm is developed to maximize the probability of recovery upon failures. Through adaptive expert placement and a flexible token dispatcher, Lazarus can also fully utilize all available nodes after failures, leaving no GPU idle. Our evaluation shows that Lazarus outperforms existing MoE training systems by up to 5.7x under frequent node failures and 3.4x on a real spot instance trace.

1.3 Dissertation Organization

In Chapter 2, we set the contexts for this dissertation. In Chapter 3, we study how manageability is currently implemented with library-based RPC solutions and discuss its problems. We present the design and implementation of mRPC to address these issues. In Chapter 4, we discuss why library based approach for collective communication is an ill fit for a multi tenant network, then we present MCCS for cloud provider managed and optimized collective communication. In Chapter 5, we discuss the challenges of running ML inference workflows on heterogeneous infrastructure, then we present our solution JellyBean. In Chapter 6, we discuss the challenges of training MoE models on spot instances, and present our resilient and elastic training system, Lazarus, to solve them. We conclude in Chapter 7.

2. Background

In this chapter, we offer background contexts for understanding the techniques proposed in this dissertation. We discuss commonly used datacenter communication primitives. We then motivate the need for manageability. Finally, we present some new types of workloads and infrastructure.

2.1 Datacenter Communication Primitives

Today's distributed applications have increasingly relied on off-the-shelf implementations of common communication primitives as core building blocks for communication between workers and different components. Two of the most widely used communication primitives are remote procedure calls (RPCs) and collective communication.

2.1.1 Remote Procedure Calls

RPC allows developers to build networked applications using a simple and familiar programming model [23], supported by several popular libraries such as gRPC [72], Thrift [257], and eRPC [112]. It enables a client application to directly call a method of a server application located on a different machine, just as if it was a local function call. The RPC model has been widely adopted in distributed data stores [114, 256, 52], network file systems [242, 67], consensus protocols [200], data-analytic frameworks [45, 253, 298, 30, 274, 7, 158, 68], cluster schedulers and orchestrators [130, 84], and machine learning systems [207, 1, 182]. Google found that roughly 10% of its datacenter CPU cycles are spent just executing gRPC library code [115]. Because of its importance, improving RPC performance has long been a major topic of research [23, 245, 18, 19, 279, 261, 276, 112, 179, 144, 38].

To use RPC, a developer defines the relevant service interfaces and message types in a schema file (e.g., gRPC `.proto` file). A protocol compiler will translate the schema into program stubs that are directly linked with the client and server applications. To issue an RPC at runtime, the application simply calls the corresponding function provided by the stub; the stub is responsible for marshalling the request arguments and interacting with the transport layer (e.g., TCP/IP sockets or RDMA verbs). The transport layer delivers the

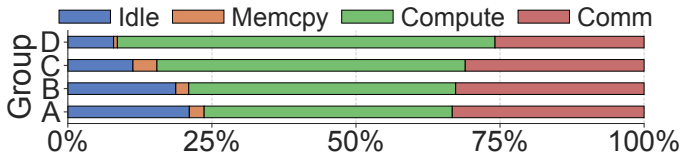


FIGURE 2.1: Training time breakdown of models from various product groups at a large social network company.

packets to the remote server, where the stub unmarshals the arguments and dispatches the RPC request to a thread (eventually replying back to the client). We refer to this approach as *RPC-as-a-library*, since all RPC functionality is included in user-space libraries that are linked with each application. Even though the first RPC implementation [23] dates back to the 1980s, modern RPC frameworks (e.g., gRPC [72], eRPC [112], Thrift [257]) still follow this same approach.

A key design goal for RPC frameworks is efficiency. Google and Facebook have built their own efficient RPC frameworks, gRPC and Apache Thrift. Although primarily focused on portability and interoperability, gRPC includes many efficiency-related features, such as supporting binary payloads. Academic researchers have studied various ways to improve RPC efficiency, including optimizing the network stack [122, 303, 201], software hardware co-design [112, 114], and overload control [38].

As network link speeds continue to scale up [219], RPC overheads are likely to become even more salient in the future. This has led some researchers to advocate for direct application access to network hardware [209, 15, 112, 303], e.g., with RDMA or DPDK. Although low overhead, kernel bypass is largely incompatible with the need for flexible and enforceable layer 7 policy control, as we discuss next. In practice, multiple security weaknesses in RDMA hardware have led most cloud vendors to opt against providing direct access to RDMA by untrusted applications [276, 166, 239, 129, 128, 308].

2.1.2 Collective Communication

Collective communication is fundamental to supporting many distributed computing workloads, especially in distributed machine learning training and inference workloads. In

distributed machine learning, collective communication operations are essential for computing and synchronizing activations and gradients across networked nodes. These communication primitives enable nodes to collaborate on shared data. For instance, AllReduce is a collective operation that aggregates data from multiple processing units, applies a chosen operator (e.g., sum), and then distributes the result back to ensure every node possesses the same global outcome.

Similar to RPCs, existing implementations for collective communication follow a library-based approach. Many popular collective communication libraries exist today, including the NVIDIA Collective Communication Library (NCCL) [189], Intel MPI library [96], OpenMPI [70], and Gloo [66]. These libraries provide common collective communication primitives such as AllReduce and AllGather, which developers leverage by linking these libraries directly into their applications. Beneath their high-level APIs, these libraries implement primitives through various algorithms along with heavily-optimized, and sometimes hardware-specific implementations. For example, NCCL provides implementations for AllReduce using both ring-based and tree-based algorithms. In Ring AllReduce, all participating GPUs form a ring structure. At each step, the i-th GPU in the ring receives a data chunk from GPU $(i-1)$ and forwards a chunk to GPU $(i+1)$. Beyond algorithm optimizations [282, 27, 249], GPUDirect RDMA for inter-host GPUs and direct GPU-to-GPU interconnects (e.g., NVLink and XGMI) for intra-host GPUs further accelerate communication. The efficiency of these collectives largely relies on algorithms that reduce network transmission.

How does a collective communication library picks a strategy for a distributed job? A collective communication library has several built-in algorithms, and the library contains logic to select one of them based on a set of static factors like data length and the number of participants. Taking AllReduce as an example, OpenMPI uses several criteria to choose the most suitable collective algorithm, including a combination of factors such as the size of the data, the number of processes involved, the network architecture, and the bandwidth and latency requirement of the algorithm [211, 81].

The performance of collective communication has recently received significant attention in the research community and industry, due to the rise of distributed deep learning. Much efforts have been place in designing algorithms to improve the performance of collective operations such as AllReduce and AllGather [232, 27, 249, 282, 203, 248], as they play critical role in deciding the end-to-end deep learning performance.

Figure 2.1 shows a statistics of contributions of to the overall training time of models across four major product groups at one of the largest social network company in the world. Exposed (non-overlapping) computation, CPU↔GPU memory copy, communication time, and GPU idle time are measured. This breakdown confirms that data communication constitutes a significant portion of the training time.

2.2 The Need for Manageability

As distributed workloads scale to large, complex deployment scenarios, there is an increasing need for improving their manageability. It is reported that Alibaba's GPU cluster needs to handle tens of thousands of training jobs each day [292]; Uber's infrastructure has 4500 microservices. Such complexity demands many manageability features. We classify the needs into three categories: 1) Observability: Provide detailed telemetry. 2) Policy Enforcement: Allow operators to apply custom policies to applications 3) Upgradability: Support software upgrades while minimizing downtime to applications.

Observability. The nature of distributed workloads implies that the end-to-end performance of an application is collectively determined by multiple nodes, including computation and communication. Observability into an application's behavior empowers both application developers and infrastructure operators to troubleshoot and optimize their applications. As communication plays an increasingly important role in modern workloads like microservices and distributed machine learning, there has been a growing interest in enhancing observability into the underlying communication substrates [314, 173, 307]. In RPCs, developers and operators want to collect metrics like latency, error rates within the traffic and overall volume of traffic. Such metrics help them identify bottlenecked components,

assess scaling decisions and prevent performance regression. In collective communication, collecting information like collective bandwidth and latency not only help operators to attribute performance regression and training failures, but also help developers to optimize their model parallelism strategies [310].

Policy Enforcement Infrastructure operator teams have sought for more control over the applications, enforcing various polices in order to restraint their behaviors, limit and regulate their performance and resource usages. For instance, in microservices, rate limits may be imposed on certain services to mitigate issues from an excessive number of RPC calls; operators may enable access control to only allow certain types of requests. In distributed machine learning, operators may enforce some quality of service polices to prioritize inference workloads over training [75]; network traffic engineering can be introduced to reduce congestion and maintain performance consistency of certain jobs [174].

Upgradability Many building blocks used by applications are frequently updated. For example, gRPC has a monthly or two-month release cycle for bug fixes and new features [73]. NCCL is also periodically updated to introduce new algorithmic optimizations and support new hardware.

2.3 New Workloads and Infrastructure

2.3.1 Machine Learning Inference Workflows

ML Inference Workflows There is a growing complexity in machine learning inference workloads both in terms of the workloads themselves as well as the computing and networking infrastructures. These workloads often involve multiple ML operators that together form a larger *ML workflow*[1]; each can be a directed acyclic graph (DAG) of ML or relational operators. For each ML operator, there are often choices of models (e.g., YOLO [235], Faster R-CNN [236]) or the same model architectures with different hyperparameters (e.g., number of layers, neural network size, choice of activation functions); inputs to the ML workflows are often collected by sensors deployed at the edge, including video cameras and an ever-expanding array of Internet-of-Things (IoT) devices. These devices may have

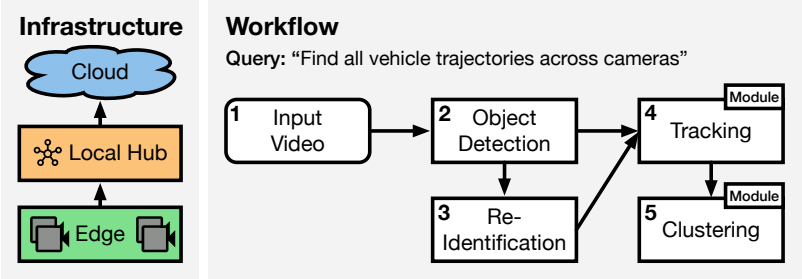


FIGURE 2.2: NVIDIA AI City Challenge for Vehicle Tracking. Some pair-wise operators are omitted for simplicity.

varying on-board compute [288] and are connected to more powerful edge-local and cloud computing services over the network.

Here we provide several examples of such ML inference workflows:

- *NVIDIA AI City challenge:* Tracking vehicles across neighboring intersections is an important ML query that allows people to understand and improve transportation efficiency [2]. The workflow is shown in Figure 2.2 with video inputs from multiple cameras of neighboring traffic intersections. It first detects objects on each individual video stream, and then performs an object re-identification (ReID) step to extract key features per detected car. A tracking module is used to find car traces in each video stream, followed by a clustering module to trace cars across different video streams.
- *Wearable health:* detecting anomalous heart signals.
- *Personal assistant:* answering complex human voice commands using Internet data.

One common characteristic is that they all rely on a set of loosely-coupled operators (i.e., operators that do not share global states but only depend on prior outputs), each of which uses an ML model or a traditional data processing module; e.g., a model to tokenize the text or relational operators such as reduce and join [21, 163]. The output of a previous operator is the input of the next, therefore formulating a workflow or logic plan in directed acyclic compute graph (DAG). Breaking down an ML query into workflows that consist of independent operators has been highly leveraged in prior research and production [250, 160]. Doing so promotes the reuse of trained models and operators to ease the development of the serving system as well as to boost performance due to shared computations [290,

135, 16, 159]; each module also can be improved independently to accelerate the application development.

2.3.2 Large Language Models

Building on the Transformer architecture, large language models (LLMs) have revolutionized the field of natural language processing as their capabilities continue to grow with the exponential scaling of model parameters. Today's LLMs far exceed the memory capacity of a single GPU, they rely on various parallelisms to distribute the model parameters and states for training and serving. For example, in data parallelism [143], the model parameters are copied on each GPU while the data is distributed among them, gradients are aggregated across all GPUs using AllReduce. In tensor parallelism [252], the model is partitioned in a way that tensors are scattered among multiple GPUs. In pipeline parallelism [188], the model is divided vertically that different GPUs compute different layers. Training LLMs have become increasingly costly, requires huge amount of GPUs for extended periods of time [172, 156]. The next generation of datacenters, with over 100,000 GPUs, costs more than $123 million annually just in electricity bills alone [246].

2.3.3 Spot Instances

Many public clouds now provide spot instances that provide significant cost savings, albeit at a price that they can be preempted at any time. For instance, AWS's spot instances are up to 70% cheaper than on-demand ones [268]. These spot instances run on sparse capacity, and may be preempted to offer capacity for on-demand instances. Frequent preemptions on spot instances pose significant challenges to utilize them for cost savings. Since preemptions are essentially "failures", without efficient failure handling and recovery, workloads that deployed on spot instances may struggle to make progress.

3. mRPC: Remote Procedure Call as a Managed System Service

As a first step towards decoupling communication, we focus on remote procedure calls (RPCs). The growing need of manageability leads to the development of sidecar based approaches to enforce polices under the library based RPC architecture. However, we find the libary and sidecar based solutions have some fundamental limitations, offering limited manageability features with significant overheads. To address this issue, in this chapter, we introduce mRPC, our alternative design where RPC is implemented as provided as a managed service.

3.1 Introduction

Recently, application and network operations teams have found a need for rapid and flexible visibility and control over the flow of RPCs in datacenters. This includes monitoring and control of the performance of specific types of RPCs [178], prioritization and rate limiting to meet application-specific performance and availability goals, dynamic insertion of advanced diagnostics to track user requests across a network of microservices [60], and application-specific load balancing to improve cache effectiveness [17], to name a few.

The typical architecture is to enforce policies in a sidecar—a separate process that mediates the network traffic of the application RPC library (Figure 3.1a). This is often referred to as a service mesh. A number of commercial products have been developed to meet the need for sidecar RPC proxies, such as Envoy [50], Istio [97], HAProxy [80], Linkerd [152], Nginx [190], and Consul [40]. Although some policies could theoretically be supported by a feature-rich RPC runtime linked in with each application, that can slow deployment—Facebook recently reported that it can take *months* to fully roll out changes to one of its application communication libraries [59]. One use case that requires rapid deployment is to respond to a new application security threat, or to diagnose and fix a critical user-visible failure. Finally, many policies are mandatory rather than discretionary—the network operations team may not be able to trust the library code linked into an application. Example

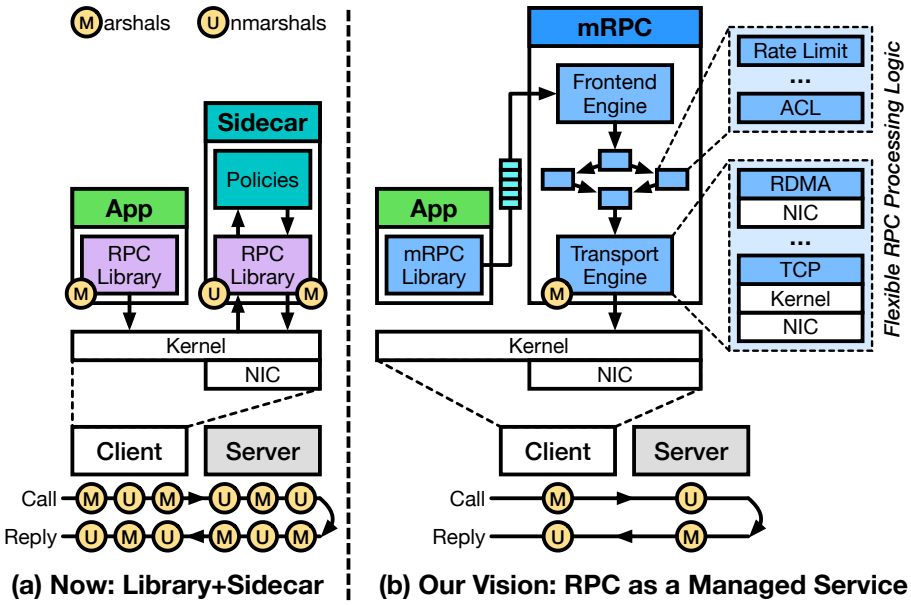


FIGURE 3.1: Architectural comparison between current (RPC library + sidecar) and our proposed (RPC as a managed service) approaches.

mandatory security policies include access control, authentication/encryption [40], and prevention of known exploits in widely used network protocols such as RDMA [239].

Although using a sidecar for policy management is functional and secure, it is also inefficient. The application RPC library marshals RPC parameters at runtime into a buffer according to the type information provided by the programmer. This buffer is sent through the operating system network stack and then forwarded back up to the sidecar, which typically needs to parse and unwrap the network, virtualization, and RPC headers, often looking inside the packet payload to correctly enforce the desired policy. It then re-marshals the data for transport. Direct application-level access to network hardware such as RDMA or DPDK offers high performance but precludes sidecar policy control. Similarly, network interface cards are increasingly sophisticated, but it is hard for applications or sidecars to take advantage of those new features, because marshalling is done too high up in the network stack. Any change to the marshalling code requires recompiling and rebooting each application and/or the sidecar, hurting end-to-end availability. In short, existing solutions can provide good performance, or flexible and enforceable policy control, but not both.

In this chapter, we propose a new approach, called RPC as a managed service, to address these limitations. Instead of separating marshalling and policy enforcement across different domains, we combine them into a single privilege and trusted system service (Figure 3.1b) so that marshalling is done after policy processing. In our prototype, mRPC for managed RPC, the privileged RPC service runs at user level communicating with the application through shared memory regions [19, 14, 166]. However, mRPC could also be integrated directly into the operating system kernel with a dynamically replaceable kernel module [176].

Our goals are to be fast, support flexible policies, and provide high availability for applications. To achieve this, we need to address several challenges. First, we need to decouple marshalling from the application RPC library. Second, we need to design a new policy enforcement mechanism to process RPCs efficiently and securely, without incurring additional marshalling overheads. Third, we need to provide a way for operators to specify/change policies and even change the underlying transport implementation without disrupting running applications.

We implement mRPC, the first RPC framework that follows the RPC as a managed service approach. Our results show that mRPC speeds up DeathStarBench [62] by up to 2.5×, in terms of mean latency, compared with combining state-of-art RPC libraries and sidecars, i.e., gRPC and Envoy, using the same transport mechanism. Larger performance gains are possible by fully exploiting network hardware capabilities from within the service. In addition, mRPC allows for live upgrades of its components while incurring negligible downtime for applications. Applications do not need to be re-compiled or rebooted to change policies or marshalling code. mRPC has three important limitations. First, data structures passed as RPC arguments must be allocated on a special shared-memory heap. Second, while we use a language-independent protocol for specifying RPC type signatures, our prototype implementation currently only works with applications written in Rust. Finally, our stub generator is not as fully featured as gRPC.

In this chapter, we make the following contributions:

- A novel RPC architecture that decouples marshalling/unmarshalling from RPC li-

braries to a centralized system service.

- An RPC mechanism that applies network policies and observability features with both security and low performance overhead, i.e., with minimal data movement and no redundant (un)marshalling. The mechanism supports live upgrade of RPC bindings, policies, transport, and marshalling without disrupting running applications.
- A prototype implementation of mRPC, along with an evaluation on both synthetic workloads and real applications.

3.2 Adding Manageability to RPC Libraries?

As the need for manageability for RPC applications grows, one natural question to ask is: *is it possible to add these properties without changing existing RPC libraries?* For observability and policy enforcement, the state-of-the-art solution is to use a sidecar (e.g., Envoy [50] or Linkerd [152]). A sidecar is a standalone process that intercepts every packet an application sends, reconstructing the application-level data (i.e., RPC), and applying policies or enabling observability. However, using a sidecar introduces substantial performance overhead, due to redundant RPC (un)marshalling. This RPC (un)marshalling, for example, in gRPC+Envoy, including HTTP framing and protobuf encoding, accounts for 62-73% overhead in the end-to-end latency [314]. In our evaluation (§3.7), using a sidecar increases the 99th percentile RPC latency by 180% and decreases the bandwidth by 44%. Figure 3.1a shows the (un)marshalling steps invoked as an RPC traverses from a client to a server and back. Using a sidecar triples the number of (un)marshalling steps (from 4 to 12). In addition, the sidecar approach is largely incompatible with the emerging trend of efficient application-level access to network hardware. Using sidecars means data buffers have to be copied between the application and sidecars, reducing the benefits of having zero-copy kernel-bypass access to the network.

Finally, using sidecars with application RPC libraries does not completely solve the upgradability issue. While policy can often be changed dynamically (depending on the feature set of the sidecar implementation), marshalling and transport code is harder to

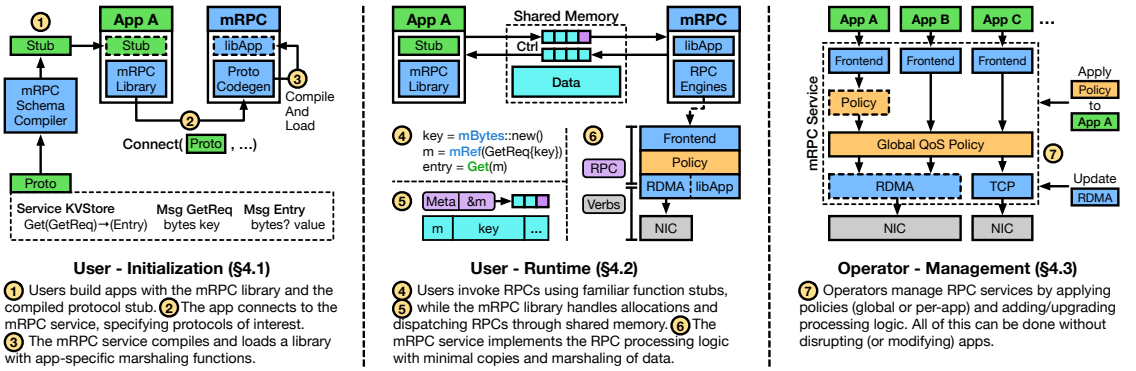


FIGURE 3.2: Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators.

change. To fix a bug in the underlying RPC library, or merely to upgrade the code to take advantage of new hardware features, we need to recompile the entire application (and sidecar) with the patched RPC library and reboot. Any scheduled downtime has to be communicated explicitly to the users of the application or has to be masked using replication; either approach can lead to complex application life-cycle management issues.

We do not see much hope in continuing to optimize this RPC library and sidecar approach for two reasons. First, a strong coupling exists between a traditional RPC library and each application. This makes upgrading the RPC library without stopping the application difficult, if not impossible. Second, there is only weak or no coupling between an RPC library and a sidecar. This prevents the RPC library and the sidecar from cross-layer optimization.

Instead, we argue for an alternative architecture in which RPC is provided *as a managed service*. By decoupling RPC logic, e.g., (un)marshalling, transport interface, from the application, the service can simultaneously provide high performance, policy flexibility, and zero-downtime upgrades.

3.3 Overview

Our system, mRPC, realizes the *RPC-as-a-managed-service* abstraction while maintaining similar end-to-end semantics as traditional RPC libraries (e.g., gRPC, Thrift). The

goals for mRPC are to be fast, support flexible policy enforcement, and provide high availability for applications.

Figure 3.2 shows a high-level overview of the mRPC architecture and workflow, breaking it down into three major phases: initialization, runtime, and management. The mRPC service runs as a non-root, user-space process with access to the necessary network devices and a shared-memory region for each application. In each of the phases, we focus on the view of a single machine that is running both the RPC client application and the mRPC service. The RPC server may also run alongside an mRPC service. In this case, mRPC-specific marshalling can be used. However, we also support flexible marshalling to enable mRPC applications to interact with external peers using well-known formats (e.g., gRPC). In our evaluation, we focus on cases where both the client and server employ mRPC.

The initialization phase extends from building the application to how the application binds to a specific RPC interface. ① Similar to gRPC, users define a protocol schema. The mRPC schema compiler uses this to generate stub code to include in their application. We illustrate this using a key-value storage service with a single `Get` function. ② When the application is deployed, it connects with the mRPC service running on the same machine and specifies the protocol(s) of interest, which are maintained by the generated stub. ③ The mRPC service also uses the protocol schema to generate, compile, and dynamically load a protocol-specific library containing the marshalling and unmarshalling code for that application's schemas[1]. This *dynamic binding* is a key enabler for mRPC to act as a long-running service, handling arbitrary applications (and their RPC schemas). [2]

At this point, we enter the runtime phase in which the application begins to invoke RPCs. Our approach uses *shared memory* between the application and mRPC, containing both control queues as well as a data buffer. ④ The application protocol stub produced by the mRPC protocol compiler can be called like a traditional RPC interface, with the exception that data structures passed as arguments or as return values must be allocated

[1] Note that such libraries may be prefetched and/or cached to optimize the startup time.

[2] The dashed box of "Stub" and "libApp" means they are generated code.

on a special heap in the shared data buffer. As an example, we show an excerpt of Rust-like pseudocode for invoking the `Get` function. ⑤ Internally, the stub and mRPC library manage RPC calls and replies in the control queues along with allocations and deallocations in the data buffer. ⑥ The mRPC service operates over the RPCs through modular *engines* that are composed to implement the per-application *datapaths* (i.e., sequence of RPC processing logic); each engine is responsible for one type of task (e.g., application interface, rate limiting, transport interface). Engines do not contain execution contexts, but are rather scheduled by *runtimes* in mRPC that correspond to kernel-level threads; during their execution, engines read from input queues, perform work, and enqueue outputs. External-facing engines (i.e., frontend, transport) use asynchronous control queues, while all other engines are executed synchronously by a runtime. Application control queues are contained in shared memory with the mRPC service.

This architecture, along with dynamic binding, enables mRPC to *operate over RPCs rather than packets*, avoiding the high overhead of traditional sidecar-based approaches. Additionally, the modular design of mRPC's processing logic enables mRPC to take advantage of fast network hardware (e.g., RDMA and smartNICs) in a manner that is transparent to the application. A key challenge, which we will address in §3.4.2, is how to securely enforce operator policies over RPCs in shared memory while minimizing data copies.

Finally, mRPC aims to improve the manageability of RPCs by infrastructure operators. Here, we zoom out to focus on the processing logic across all applications served by an mRPC service. ⑦ Operators may wish to apply a number of different policies to RPCs made by applications, whether on an individual basis (e.g., rate limiting, access control) or globally across applications (e.g., QoS). mRPC allows operators to add, remove, update, or reconfigure policies at runtime. This flexibility extends beyond policies to include those responsible for interacting with the network hardware. A key challenge, which we will address in §3.4.3, is in supporting the *live upgrade* of mRPC engines without interrupting running applications (and while managing engines sharing memory queues).

3.4 Design

In this section, we describe how mRPC provides dynamic binding, efficient policy and observability support, live upgrade, and security.

3.4.1 Dynamic RPC Binding

Applications have different RPC schemas, which ultimately decide how an RPC is marshalled. In the traditional RPC-as-a-library approach, a protocol compiler generates the marshalling code, which is linked into the application. In our design, the mRPC service is responsible for marshalling, which means that the application-specific marshalling code needs to be decoupled from an RPC library and run inside the mRPC service itself. Failing to ensure this separation would allow arbitrary code execution by a malicious user.

Applications directly submit the RPC schema (and not marshalling code) to the mRPC service. The mRPC service generates the corresponding marshalling code, then compiles and dynamically loads the library. Thus, we rely on our mRPC service code generator to produce the correct marshalling code for *any* user-provided RPC schema. For the initial handshake between an RPC client and an RPC server, the two mRPC services check that the provided RPC schemas match, and if not, the client's connection is rejected.

There are three remaining questions. First, *what are the responsibilities of the in-application user stub and mRPC library?* In mRPC, applications rely on user stubs to implement the abstraction as specified in their RPC schema. This means we still need to generate the glue code to maintain the traditional application programming interface. Our solution is to provide a separate protocol schema compiler, which is untrusted and run by application developers, to generate the user stub code that does not involve marshalling and transport. The application RPC stub (with the help of the mRPC library) creates a message buffer that contains the metadata of the RPC, with typed pointers to the RPC arguments, on the shared memory heap. The message is placed on a shared memory queue, which will be processed by the mRPC service. The receiving side works in a similar way.

Second, *does this approach increase RPC connect/bind time?* Implemented naively,

this design will increase the RPC connect/bind time because the mRPC service has to compile the RPC schema and load the resulting marshalling library when an RPC client first connects to a corresponding server (or equivalently when an RPC server binds to the service). However, this latency is not fundamental to our design, and we can mitigate it in the following way. The mRPC service accepts RPC schemas before booting an application, as a form of prefetching. Given a schema, it compiles and caches the marshalling code. At the time of RPC connect/bind, the mRPC service simply performs a cache lookup based on the hash of the RPC schema. If it exists within the cache, the mRPC service will load the associated library; otherwise, the mRPC service will invoke the compiler to generate (and subsequently cache) the library. This reduces the connect/bind time from several seconds to several milliseconds.

Third, *when new applications arrive, do existing applications face downtime?* The multi-threaded mRPC service is a single process that serves many RPC applications; however, the marshalling engines for different RPC applications are not shared. They are in different memory addresses and can be (un)loaded independently. We will describe in §3.4.3 how to load/unload engines without disrupting running applications.

3.4.2 Efficient RPC Policy Enforcement and Observability

We have one key idea to allow efficient RPC policy enforcement and observability: senders should marshal once (as late as possible), while receivers should unmarshal once (as early as possible). On the sender side, we want to support policy enforcement and observability directly over RPCs from the application, and then marshal the RPC into packets. The receiver side is similar: packets should be unmarshalled into RPCs, applying policy and observability operations, and then delivered directly to the application. Compared to the traditional RPC-as-a-library approach with sidecars, this eliminates the redundant (un)marshalling steps (see Figure 3.1).

Data: DMA-capable shared memory heaps. Our design is centered around a dedicated shared memory heap between each application and the mRPC service. (Note that this heap

is not shared across applications.) Applications directly create data structures, which may be used in RPC arguments, in a shared memory heap with the help of the mRPC library. Each application has a separate shared memory region, which provides isolation between (potentially mutually distrusting) applications. The mRPC library also includes a standard slab allocator for managing object allocation on this shared memory. If there is insufficient space within the shared memory, the slab allocator will request additional shared memory from the mRPC service and then map it into the application's address space. The mRPC service has access to the shared memory heap, allowing it to execute RPC processing logic over the application's RPCs, but also maintains a private memory heap for necessary copies.

Figure 3.3 shows an example workflow that includes access control for a key-value store service. Having the data structures directly in the shared memory allows an application to provide pointers to data, rather than the data itself, when submitting RPCs to the mRPC service. We call the message sent from an application to the mRPC service an *RPC descriptor*. If there are multiple RPC arguments, the RPC descriptor points to an array of pointers (each pointing to a different argument on the heap).

Let us say we have an ACL policy that rejects an RPC if the key matches a certain string. The mRPC service first copies the argument (i.e., `key`), as well as all parental data structures (i.e., `GetReq`), onto its private heap. This is to prevent time-of-use-to-time-of-check (TOCTOU) attacks. Since applications have access to DMA-capable shared memory at all times, an application could modify the content in the memory while the mRPC service is enforcing policies. Copying arguments is a standard mitigation technique, similar to how OS kernels prevents TOCTOU attacks by copying system call arguments from user- to kernel-space. This copying only needs to happen if the policy behavior is based on the content of the RPC. We demonstrate in §3.7.2 that even with such copying, mRPC's overhead for an ACL policy is much lower than gRPC + Envoy. The RPC descriptor is modified so that the pointer to the copied argument now points to the private heap. On the receiver side, the TOCTOU attack is not relevant, but we need to take care not to place RPCs directly in shared memory. If there is a receive-side policy that depends on RPC

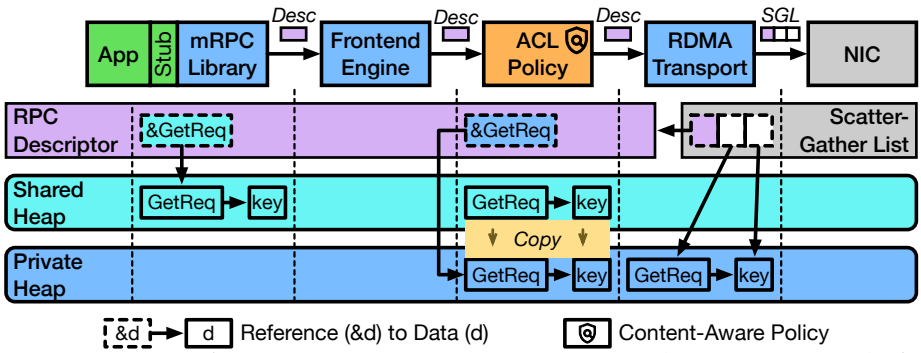


FIGURE 3.3: Overview of memory management in mRPC. Shows an example for the `Get` RPC that includes a content-aware ACL policy.

argument values, the mRPC service first receives the RPC data into a private heap; it copies the RPC data into the shared heap after policy processing. This prevents the application from reading RPC data that should have been dropped or modified by the policies. Note that we can bypass this copy when processing does not depend on RPC argument values (e.g., rate limits). During ACL policy enforcement, the RPC is dropped if the `key` argument is contained in a blocklist. Note that if an RPC is dropped, any further processing logic is never executed (including marshalling operations).

Finally, at the end of the processing logic, the transport adapter engine executes. mRPC currently supports two types of transport: TCP and RDMA. For TCP, mRPC uses the standard, kernel-provided scatter-gather (`iovec`) socket interface. For RDMA, mRPC uses the scatter-gather `verb` interface, allowing the NIC to directly interact with buffers on the shared (or private) memory heaps containing the RPC metadata and arguments. For both TCP and RDMA, mRPC provides disjoint memory blocks to the transport layer directly, eliminating excessive data movements.[3]

Control: Shared-memory queues. To facilitate efficient communication between an application and the mRPC service, we use shared memory control queues. mRPC allocates two unidirectional queues for sending and receiving requests from an application to the

[3] For RDMA, if the number of disjoint memory blocks exceeds the limit of NIC's capability to encapsulate all blocks in one RDMA work request, mRPC coalesces the data into a memory block before transmission. This is because sending a single work request (even with a copy) is faster than sending multiple smaller work requests on our hardware.

mRPC service. The requests contain RPC descriptors, which reference arguments on the shared memory heap. The mRPC service always copies the RPC descriptors applications put in the sending queue to prevent TOCTOU attacks. mRPC provides two options to poll the queues: 1) busy polling, and 2) `eventfd`-based adaptive polling. In busy polling, both the application-side mRPC library and the mRPC service busy poll on their ends of the queues. In the `eventfd` approach, the mRPC library and the mRPC service sends event notifications after enqueuing to an empty queue. After receiving a notification, the queue is drained (performing the necessary work) before subsequently waiting on future events. The `eventfd` approach saves CPU cycles when queues are empty. Other alternative solutions may involve dynamically scaling up (or down) the number of threads used to busy poll by the mRPC service; however, we chose the `eventfd` approach for its simplicity. In our evaluation, we use busy polling for RDMA and eventfd-based adaptive polling for TCP.

Memory management. We provide a memory allocator in the mRPC library for applications to directly allocate RPC data structures to be sent on a shared memory heap. The allocator invokes the mRPC service to allocate shared memory regions on behalf of the application (similar to how a standard heap manager calls `mmap` or `sbrk` to allocate memory from an OS kernel). We need to use a specialized memory allocator for RPC messages (and their arguments), since RPCs are shared between three entities: the application, the mRPC service, and the NIC. A memory block is safe to be reclaimed only when it will no longer be accessed by any entity.

We adopt a notification-based mechanism for memory management. On the sender side, the outgoing messages are managed by the mRPC library within the application. On the receiver side, the incoming messages are managed by the mRPC service. When the application no longer accesses a memory block occupied by outgoing messages, the memory block will not be reclaimed until the library receives a notification from mRPC service that the corresponding messages are already sent successfully through the NIC (similar to how zero-copy sockets work in Linux). Incoming messages are put in buffers

on a separate read-only shared heap. The receiving buffers can be reclaimed when the application finishes processing (e.g., when the RPC returns). To support reclamation of receive buffers, the mRPC library notifies the mRPC service when specific messages are no longer in use by the application. Notifications for multiple RPC messages are batched to improve performance. If the receiver application code wishes to preserve or modify the incoming data, it must make an explicit copy. Although this differs from traditional RPC semantics, in our implementation of Masstree and DeathStarBench we found no examples where the extra copy was necessary.

Cross-datapath policy engines. mRPC supports engines that operate over multiple datapaths, which may span multiple applications. For instance, any global policy (e.g., QoS) will need to operate over all datapaths (see §3.5). For this type of engine, we instantiate replicas of the engine for each datapath that it applies to. Replicas can choose to either communicate through shared state, which requires managing contention across runtimes, or support runtime-local state that is contention-free.

3.4.3 Live Upgrades

Although our modular engine design for the mRPC service is similar to Snap [166] and Click [127], we arrive at very different designs for upgrades. Click does not support live upgrades, while Snap executes the upgraded process to run alongside the old process. The old process serializes the engine states, transfers them to the new process, and the new process restarts them. This means that even changing a single line of code within a single Snap engine requires a complete restart for all Snap engines. This design philosophy is fundamentally not compatible with mRPC, as we need to deal with new applications arriving with different RPC schemas, and thus our upgrades are more frequent. In addition, we want to avoid fate sharing for applications: changes to an application's datapath should not impact the performance of other applications. Ultimately, Snap is a network stack that does not contain application-specific code, where as mRPC needs to be application-aware for marshalling RPCs.

We implement engines as plug-in modules that are dynamically loadable libraries. We design a live upgrade method that supports *upgrading, adding, or removing components of the datapath without disrupting other datapaths.*

Upgrading an engine. To upgrade one engine, mRPC first detaches the engine from its runtime (preventing it from being scheduled). Next, mRPC destroys and deallocates the old engine, but maintains the old engine's state in memory; note that the engine is detached from its queues and not running at this time. Afterwards, mRPC loads the new engine and configures its send and receive queues. The new engine starts with the old engine's state. If there is a change in the data structures of the engine's state, the upgraded engine is responsible for transforming the state as necessary (which the engine developer must implement). Note that this also applies to any shared state for cross-datapath engines. The last step is for mRPC to attach the new engine to the runtime.

Changing the datapath. When an operator changes the datapath to add or remove an engine, this process now involves the creation (or destruction) of queues and management of in-flight RPCs. Changes that add an engine are straightforward, since it only involves detaching and reconfiguring the queues between engines. Changes that remove an engine are more complex, as some in-flight RPCs may be maintained in internal buffers; for example, a rate limiter policy engine maintains an internal queue to ensure that the output queue meets a configured rate. Engine developers are responsible for flushing such internal buffers to the output queues when the engines are removed.

Multi-host upgrades or datapath changes. Some engine upgrades or datapath changes that involve both the sender and the receiver hosts need to carefully manage in-flight RPCs across hosts. For example, if we want to upgrade how mRPC uses RDMA, both the sender and the receiver have to be upgraded. In this scenario, the operator has to develop an upgrade plan that may involve upgrading an existing engine to some intermediate, backward-compatible engine implementation. The plan also needs to contain the upgrade sequence, e.g., upgrading the receiver side before the sender side. Our evaluation demonstrates such

a complex live upgrade, which optimizes the handling of many small RPC requests over RDMA (see §3.7.3).

3.4.4 Security Considerations

We envision two deployment models for mRPC: (1) a cloud tenant uses mRPC to manage its RPC workloads (similar to how sidecars are used today); (2) a cloud provider uses mRPC to manage RPC workloads on behalf of tenants. In both models, there are two different classes of principals: operators and applications. Operators are responsible for configuring the hardware/virtual infrastructure, deploying the mRPC service, and setting up policies that mRPC will enforce. Applications run on an operator's infrastructure, interacting with the mRPC service to invoke RPCs. Applications trust operators, along with all privileged software (e.g., OS) and hardware that the operators provide; both applications and operators trust our mRPC service and protocol compiler. In both deployment models, applications are not trusted and may be malicious (e.g., attempt to circumvent network policies).

In the first deployment model, mRPC service runs on top of a virtualized network that is dedicated to the tenant. Running arbitrary policy and observability code inside the mRPC service cannot attack other tenants' traffic since inter-tenant isolation is provided by the cloud provider. In the second deployment model, our current prototype does not support running tenant-provided policy implementation inside mRPC service. How to safely integrate tenant-provided policy implementation and a cloud provider's own policy implementation is a future work.

From the application point of view, we want to ensure that mRPC provides equivalent security guarantees as compared to today's RPC library and sidecar approach, which we discuss in terms of: 1) dynamic binding and 2) policy enforcement. Our dynamic binding approach involves the generation, compilation, and runtime loading of a shared library for (un)marshalling application RPCs. Given that the compiled code is based on the application-provided RPC schema, this is a possible vector of attack. The mRPC schema

compiler is trusted with a minimal interface: other than providing the RPC schema, applications have no control on the process of how the marshalling code is generated. We open source our implementation of the compiler so that it can be publicly reviewed.

As for all of our RPC processing logic, policies are enforced over RPCs by operating over their representations in shared memory control queues and data buffers. With a naive shared memory implementation, this introduces a vector of attack by exploiting a time-of-check to time-of-use (TOCTOU) attack; for instance, the application could modify the RPC message after policy enforcement but before the transport engine handles it. In mRPC, we address this by copying data into an mRPC-private heap prior to executing any policy that operates over the content of an RPC (as opposed to metadata such as the length). Similarly, received RPCs cannot be placed in shared memory until all policies have been enforced, since otherwise applications could see received RPCs before policies have a chance to drop (or modify) them. Shared memory regions are maintained by the mRPC service on a per-application basis to provide isolation.

3.5 Advanced Manageability Features

mRPC's architecture creates an opportunity for advanced manageability features such as cross-application RPC scheduling. In this section, we present two such features that we developed on our policy engine framework to demonstrate the broader utility of our RPC-as-a-managed-service architecture.

Feature 1: Global RPC QoS. mRPC allows centralized RPC scheduling of cross-application workloads based on a global view of current outstanding RPCs. For example, mRPC can enforce a policy that prioritizes RPCs with earliest deadlines [260] across applications to support latency SLO or prioritizes latency-sensitive workloads [308]. One challenge here is that a naive implementation may attempt to apply the QoS policy for datapaths spread over multiple runtimes (i.e., execution thread contexts). This would require the (replicated) policy engines on each datapath to share the state on outstanding RPCs, and thus impose synchronization overheads. Therefore, we adopt a similar strategy as used in the Linux

kernel to apply the QoS policy on a per-runtime basis, which instead can use runtime-local storage without the need for synchronization. In our implementation, we support a QoS strategy that prioritizes small RPCs based on a configurable threshold size.

Feature 2: Avoiding RDMA performance anomalies. It is well known that RDMA workloads may not fully utilize the capability of a specific RDMA NIC without fine-tuning, and that particular traffic patterns can even cause performance anomalies [113, 128] (e.g., low RDMA throughput, pause frame storms). Previous work such as ScaleRPC [35] and Flock [179] have proposed techniques to utilize the RNIC more efficiently. However, their approaches are library-based and only work for single applications; therefore, they do not handle scenarios in which the *combination* of multiple application workloads causes poor RDMA performance. mRPC's architecture enables us to have a global view of all RDMA requests and to avoid such performance anomalies.

We implement a global RDMA scheduler inside the RDMA transport engine, which translates RPC requests into RDMA messages and sends them to the RDMA NIC. In our implementation, we focus on addressing the performance degradation from interspersed small and large scatter-gather elements (which may be across RPCs as well as applications). We fuse such elements together with an explicit copy with an upper bound of 16 KB for the size of the fused element.

3.6 Implementation

mRPC is implemented in 32K lines of Rust: 3K lines for the protocol compiler, 6K for the mRPC control plane, 12K for engine implementations, and 11K for the mRPC library. The mRPC control plane is part of the mRPC service that loads/unloads engines.

The mRPC control plane is not live-upgradable. The mRPC library is linked into applications and is thus also not live-upgradable. We do not envision the need to frequently upgrade these components because they only implement the high-level, stable APIs, such as shared memory queue communication and (un)loading engines.

Engine interface. Table 3.1 presents the essential API functions that all engines must

Table 3.1: mRPC Engine Interface.

Operations
`doWork`(in:[Queue], out:[Queue]) *Operate over one or more RPCs available on input queues.*
`decompose`(out:[Queue]) → State *Decompose the engine to its compositional states.* *(Optionally output any buffered RPCs)*
`restore`(State) → Engine *Restore the engine from the previously decomposed state.*

implement. Each engine represents some asynchronous computation that operates over input and output queues via `doWork`, which is similar in nature to Rust's `Future`. mRPC uses a pool of runtime executors to drive the engines by calling `doWork`, where each runtime executor corresponds to a kernel thread. We currently implement a simple scheduling strategy inspired by Snap [166]: engines can be scheduled to a dedicated or shared runtime on start. In addition, runtimes with no active engines will be put to slept and release CPU cycles. The engines also implement APIs to support live upgrading: `decompose` and `restore`. In `decompose`, the engine implementation is responsible for destructing the engine and creating a representation of the final state of the engine in memory, returning a reference to mRPC. mRPC invokes `restore` on the upgraded instance of the engine, passing in a reference to the final state of the old engine. The developer is responsible for handling backward compatibility across engine versions, similar to how application databases may be upgraded across changes to their schemas.

Transport engines. We abstract reliable network communication of messages into transport engines, which share similar design philosophy with Snap [166] and TAS [122]. We currently implement two transport engines: RDMA and TCP. Our RDMA transport engine is implemented based on OFED libibverbs 5.4, while our TCP transport engine is built on Linux kernel's TCP socket.

mRPC Library. Modern RPC libraries allow the user to specify the RPC data types and service interface through a language-independent schema file (e.g., `protobuf` for gRPC,

`thrift` for Apache Thrift). mRPC implements support for `protobuf` and adopts similar service definitions as gRPC, except for gRPC's streaming API. mRPC also integrates with Rust's async/await ecosystem for ease of asynchronous programming in application development.

To create an RPC service, the developer only needs to implement the functions declared in the RPC schema. The dependent RPC data types are automatically generated and linked with the application by the mRPC schema compiler. The mRPC library handles all the rest, including task dispatching, thread management, and error handling. To allow applications to directly allocate data in shared memory without changing the programming abstraction, we implement a set of shared memory data structures that expose the same rich API as Rust's standard library. This is done by replacing the memory allocation of data structures such as `Vec` and `String` with the shared memory heap allocator.

3.7 Evaluation

We evaluate mRPC using an on-premise testbed of servers with two 100 Gbps Mellanox Connect-X5 RoCE NICs and two Intel 10-core Xeon Gold 5215 CPUs (running at 2.5 GHz base frequency). The machines are connected via a 100 Gbps Mellanox SN2100 switch. Unless specified otherwise, we keep a single in-flight RPC to evaluate latency. To benchmark goodput and RPC rate, we let each client thread keep 128 concurrent RPCs on TCP and 32 concurrent RPCs on RDMA.

3.7.1 Microbenchmarks

We first evaluate mRPC's performance through a set of microbenchmarks over two machines, one for the client and the other for the server. The RPC request has a byte-array argument, and the response is also a byte array. We adjust the RPC size by changing the array length. RPC responses are an 8-byte array filled with random bytes. We compare mRPC with two state-of-the-art RPC implementations, eRPC and gRPC (v1.48.0). We deploy Envoy (v1.20) in HTTP mode to serve as a sidecar for gRPC. We use mRPC's TCP and RDMA backends to compare with gRPC and eRPC, respectively. There is no existing

sidecar that supports RDMA. To evaluate the performance of using a sidecar to control eRPC traffic, we implement a single-thread sidecar proxy using the eRPC interface. We keep applications running for 15 seconds to measure the result.

Small RPC latency. We evaluate mRPC's latency by issuing 64-byte RPC requests over a single connection. Table 3.2 shows the latency for small RPC requests. Note that since the marshalling of small messages is fast on modern CPUs, the result in the table remains stable even when the message size scales up to 1 KB. We use `netperf` and `ib_read_lat` to measure raw round-trip latency.

mRPC achieves median latency of 32.8 µs for TCP and 7.6 µs for RDMA. Relative to netperf (TCP) or a raw RDMA read, mRPC adds 11.8 or 5.1 µs to the round-trip latency. This is the cost of the mRPC abstraction on top of the raw transport interface (e.g., socket, verbs).

We also evaluate latency in the presence of sidecar proxies. The sidecars do not enforce any policies, so we are only measuring the base overhead. Our results show that adding sidecars substantially increases the RPC latency. On gRPC, adding Envoy sidecars more than triples the median latency. The result is similar with eRPC. On mRPC, having a NullPolicy engine (which simply forwards RPCs) in the mRPC service has almost no effect on latency, increasing the median latency only by 300 ns.

Comparing the full solution (mRPC with policy versus gRPC/eRPC with proxy), mRPC speeds up the median latency by 6.1× (i.e., 33.4 µs against 203.4 µs) and the 99th percentile tail latency by 5.8×. On RDMA, mRPC speeds up eRPC by 1.3× and 1.4× in terms of median and tail latency (respectively). This is because the communication between the eRPC app and its proxy goes through the NIC, which triples the cost in the end-host driver (including the PCIe latency). In contrast, mRPC's architecture shortcuts this step with shared memory.

In addition, to separate the performance gain from system implementation difference, we evaluate the latency of mRPC with full gRPC-style marshalling (protobuf encoding and

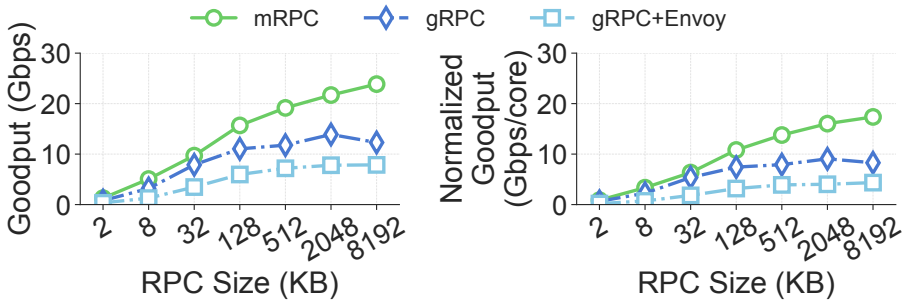
Table 3.2: Microbenchmark [Small RPC latency]: Round-trip RPC latencies for 64-byte requests and 8-byte responses.

Transport	Solution	Median Latency (µs)	P99 Latency (µs)
TCP	Netperf	21.0	32.0
	gRPC	63.0	90.3
	mRPC	32.8	38.7
	gRPC+Envoy	203.4	251.1
	mRPC+NullPolicy	**33.4**	**43.3**
	mRPC+NullPolicy+HTTP+PB	49.8	61.9
RDMA	RDMA read	2.5	2.8
	eRPC	3.6	4.1
	mRPC	7.6	8.7
	eRPC+Proxy	11.3	15.6
	mRPC+NullPolicy	**7.9**	**9.1**

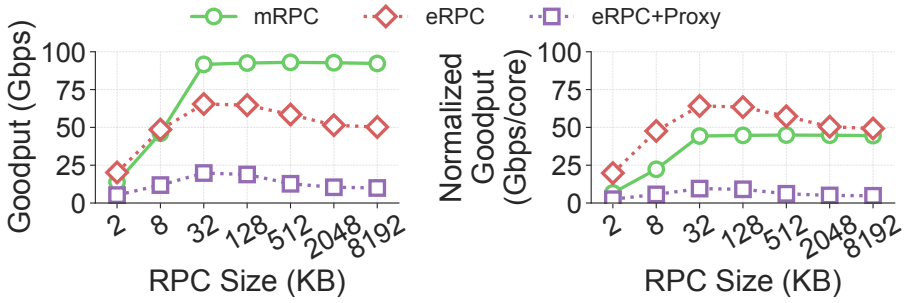
HTTP/2 framing) in the presence of NullPolicy engines as an ablation study. Under this setting, compared with gRPC + Envoy, mRPC speeds up the latency by 4.1× in terms of both median and tail latency. We also observe that the mRPC framework does not introduce significant overhead. Even with the cost of protobuf and HTTP/2 encoding, mRPC still achieves slightly lower latency compared with standalone gRPC. In mRPC, we can choose a customized marshalling format, because we know the other side is also an mRPC service. In other cases, e.g., when interfacing with external traffic or dealing with endianness differences, we can still apply full-gRPC style marshalling. When mRPC is configured to use full-gRPC style marshalling, we only need to pay (un)marshalling costs between mRPC services. For gRPC + Envoy, in addition to the (un)marshalling costs between Envoy proxies, the communication between applications and Envoy proxies also needs to pay this (un)marshalling cost. In the remaining evaluations, we will use mRPC's customized marshalling protocol. More results using gRPC-style marshalling are shown in Appendix A.

Large RPC goodput. The client and server in our goodput test use a single application thread. The left side of Figure 3.4 shows the result. From this point on, when we discuss mRPC's performance, we focus on the performance of mRPC that has at least a NullPolicy engine in place to fairly compare with sidecar-based approaches.

mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.1× and 9.3×, respec-



(a) TCP-based transport

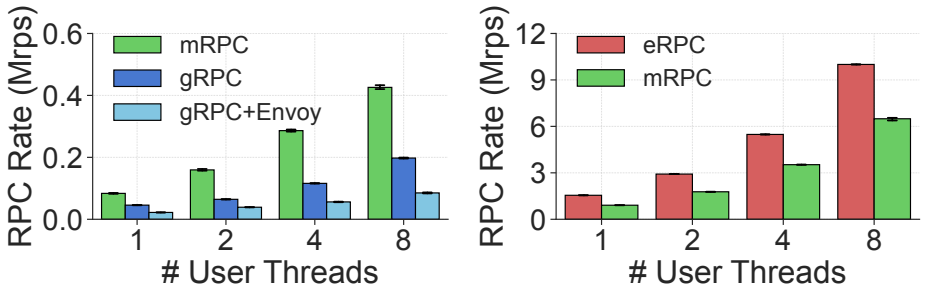


(b) RDMA-based transport

FIGURE 3.4: Microbenchmark [Large RPC goodput]: Comparison of goodput for large RPCs. Note that different solutions demand different amounts of CPU cores, so we also normalized the goodput to their CPU utilization, as shown in the right figures. The error bars show the 95% confidence interval, but they are too small to be visible.

tively, for 8KB RPC requests. mRPC is especially efficient for large RPCs[4], for which (un)marshalling takes a higher fraction of CPU cycles in the end-to-end RPC datapath. Having a sidecar substantially hurts RPC goodput both for TCP and RDMA. In particular, for RDMA, intra-host roundtrip traffic through the RNIC might contend with inter-host traffic in the RNIC/PCIe bus, halving the available bandwidth for inter-host traffic. mRPC even outperforms gRPC (without Envoy). mRPC is fundamentally more efficient in terms of marshalling format: mRPC uses `iovec` and incurs no data movement. Appendix A shows an ablation study that demonstrates that even if mRPC uses a full gRPC-style marshalling engine, mRPC outperforms gRPC + Envoy due to a reduction in the number of (un)marshalling steps.

CPU overheads. To understand the mRPC CPU overheads, we measure the per-core



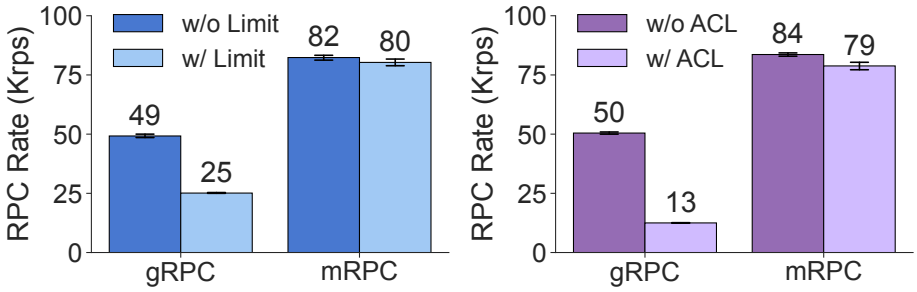
(a) TCP-based transport (b) RDMA-based transport

FIGURE 3.5: Microbenchmark [RPC rate and scalability]: Comparison of small RPC rate and CPU scalability. The bars show the RPC rate.

goodput. The results are shown on the right side of Figure 3.4. mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.8× and 9.3×, respectively. This means mRPC is much more CPU-efficient than gRPC + Envoy and eRPC + Proxy. eRPC (without a proxy) is quite efficient, but converges to mRPC's efficiency as RPC size increases.

RPC rate and scalability. We evaluate mRPC's small RPC rate and its multicore scalability. We fix the RPC request size to 32 bytes and scale the number of client threads. We use the same number of threads for the server as the client, and each client connects to one server thread. Figure 3.5 shows the RPC rates when scaling from 1 to 8 user threads, where the error bars show the 95% confidence interval. All the tested solutions scale well. mRPC's RPC rates scale by 5.1× and 7.2×, on TCP and RDMA, from a single thread to 8 threads. As a reference, gRPC scales by 4.3×, gRPC + Envoy scales by 3.9×, and eRPC scales by 6.5×. mRPC achieves 0.43 Mrps on TCP and 6.5 Mrps on RDMA with 8 threads. gRPC + Envoy only has 0.09 Mrps, so mRPC outperforms it by 5×. We do not evaluate eRPC + proxy, because our eRPC proxy is only single-threaded. When we run eRPC + proxy with a single thread, it achieves 0.51 Mrps. So even if eRPC + proxy scales linearly to 8 threads, mRPC still outperforms it.

[4] Standalone eRPC exhibits relatively lower goodput on RoCE than on Infiniband. According to the eRPC paper [112], eRPC should achieve 75 Gbps on Infiniband for 8MB RPCs.



(a) Rate Limiting (b) Access Control

FIGURE 3.6: Efficient Support for Network Policies. The RPC rates with and without policy are compared.

3.7.2 Efficient Policy Enforcement

We use two network policies as examples to demonstrate mRPC's efficient support for RPC policies: (1) RPC rate limiting and (2) access control based on RPC arguments. RPC rate limiting allows an operator to specify how many RPCs a client can send per second. We implement rate limiting as an engine using the token bucket algorithm [266]. Our access control policy inspects RPC arguments and drops RPCs based on a set of rules specified by network operators. These two network policies differ greatly from traditional rate limiting and access control, which only limit network bandwidth and can only operate on packet headers.

We compare rate limit enforcement using an mRPC policy versus using Envoy's rate limiter on gRPC workloads. To evaluate the performance overheads, we set the limit to infinity so that the actual RPC rate is never above the limit (allowing us to observe the overheads). Figure 3.6a shows the RPC rate with and without the rate limits. The bars of w/o Limit for gRPC show its throughput when the sidecar is bypassed. The error bars show the 95% confidence interface. gRPC's RPC rate drops immediately from 49K to 25K. This is because having a sidecar proxy (Envoy) introduces substantial performance overheads. For mRPC, the RPC rate stays the same at 82K. This is because having a policy introduces minimal overheads. The extra policy only adds tens to hundreds of extra CPU instructions on the RPC datapath.

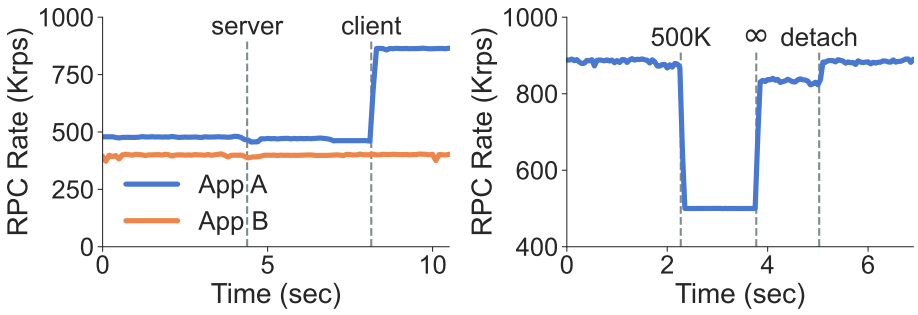
We evaluate access control on a hotel reservation application in DeathStarBench [62]. The service handles hotel reservation RPC requests, which include the customer's name, the check-in date, and other arguments. The service then returns a list of recommended hotel names. We set the access control policy to filter RPCs based on the `customerName` argument in the request. We use a synthetic workload containing 99% valid and 1% invalid requests. We again compare our mRPC policy against using Envoy to filter gRPC requests. We implement the Envoy policy using WebAssembly. gRPC's rate drops from 50K to 13K. This is because of the same sidecar overheads and now Envoy has to further parse the packets to fetch the RPC arguments. On mRPC, the performance drop is much smaller, from 84K to 79K. Note that, on mRPC, the performance overhead of introducing access control is larger than rate limiting. For access control, the mRPC service has to copy the relevant field (i.e., `customerName`) to the private heap to prevent TOCTOU attacks on the sender side and has to copy the RPC from a private heap to the shared heap on the receiver side.

3.7.3 Live Upgrade

We demonstrate mRPC's ability to live upgrade using two scenarios.

Scenario 1. During our development of mRPC, we realized that using the RDMA NIC's scatter-gather list to send multiple arguments in a single RPC can significantly boost mRPC's performance. In this approach, even when an RPC contains arguments that are scattered in virtual memory, we can send the RPC using a single RDMA operation (`ibv_post_send`). We use these two versions of our RDMA transport engine to demonstrate that mRPC enables such an upgrade without affecting running applications. Note that all other evaluations already include this RDMA feature. This upgrade involves both the client side's mRPC service and the server side's mRPC service, because it involves how RDMA is used between machines (i.e., transport adapter engine). gRPC and eRPC cannot support this type of live upgrade.

We run two applications (App A and App B). Both applications are sending 32-byte



(a) Transport adapter (b) Rate limit policy

FIGURE 3.7: Live upgrade. The annotations in (a) indicate when A's client and A and B's servers are upgraded; in (b) the specified rate and when the policy is removed.

RPCs, and the responses are 8 bytes. A and B share the mRPC service on the server side. A's and B's RPC clients are on different machines. We keep 8 concurrent RPCs for B, forcing it to send at a slower rate, while using 32 for A. We first upgrade the server side to accept arguments as a scatter-gather list, and we then upgrade the client side of A. Figure 3.7a shows the RPC rate of A and B. When the server side upgrades, we observe a negligible effect on A's and B's rate. Neither A nor B needs recompilation or rebooting. When A's client side's mRPC service is upgraded, A's performance increases from 480K to 860K. B's performance is not affected at all because B's client side's mRPC service is not upgraded.

Scenario 2. Enforcing network policies has performance overheads, even when they do not have any effect. For example, enforcing a rate limit of an extremely large throttle rate still introduces performance overheads just for tracking the current rate using token buckets. mRPC allows policies to be removed at runtime, without disrupting running applications.

We use the same rate limiting setup from §3.7.2 but on top of RDMA transport. Figure 3.7b shows the RPC rate. We start from not having the rate limit engine. We then load the rate limit engine and set the throttled rate to 500K. The RPC rate immediately becomes 500K. We then set the throttled rate to be infinite, and the rate becomes 840K. After we detach the rate limit engine, the rate becomes 890K.

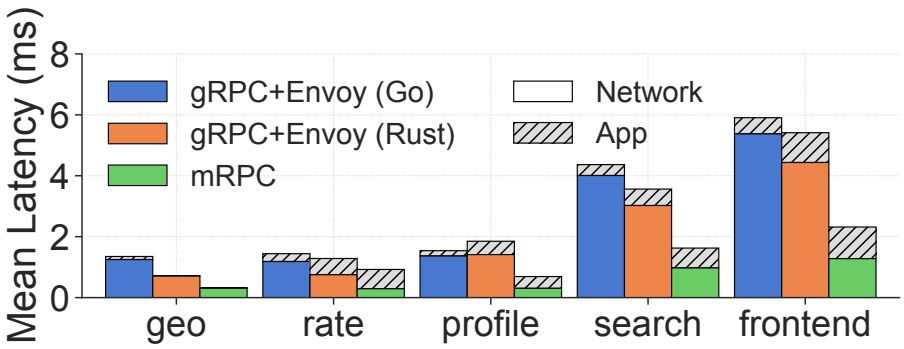


FIGURE 3.8: DeathStarBench: Mean latency of in-app processing and network processing of microservices. The latency of a microservice includes RPC calls to others.

Takeaways. There are two overall takeaways from these experiments. First, mRPC allows upgrades to the mRPC service without disrupting running applications. Second, live upgrades allow for more flexible management of RPC services, which can be used to enable immediate performance improvements (without redeploying applications) or dynamic configuration of policies.

3.7.4 Real Applications

We evaluate how the performance benefits of mRPC transform into end-to-end application-level performance metrics.

DeathStarBench. We use the hotel reservation service from the DeathStarBench [62] microservice benchmark suite. The reference benchmark is implemented in Go with gRPC and Consul [40] (for service discovery). Our mRPC prototype currently only supports Rust applications, and we thus port the application code to Rust for comparison. We use the same open-source services such as memcached [168] and MongoDB [180].

We distribute the HTTP frontend and the microservices on four servers in our testbed. The monolithic services (memcached, MongoDB) are co-located with the microservices that depend on them. We use a single thread for each of the microservices and the frontend. Further, we deploy an Envoy proxy as a sidecar on each of the servers (with no active policy). The provided workload generator [62] is used to submit HTTP requests to the frontend. For a fair comparison, we also implemented a Rust version of the benchmark

Table 3.3: Masstree analytics: Latency and the achieved throughput for GET operations. MOPS is Million Operations Per Second.

	Median Latency	P99 Latency	Throughput
eRPC	16.8 µs	21.7 µs	8.7 MOPS
mRPC	22.5 µs	33.1 µs	7.0 MOPS

with Tonic [272], which is the de facto implementation of gRPC in Rust. We deploy the mRPC and Tonic implementations on bare metal, while the reference Go suite runs in Docker containers with a host network (which introduces negligible performance overheads compared to using bare metal [316]). All three solutions are based on TCP. We issue 20 requests per second for 250 seconds and record the latency of each request, breaking it down into the in-application processing time and network processing time for each microservice involved. In our evaluation, the dynamic bindings of the user applications are already cached in mRPC service, so the time to generate the bindings is not included in the result.

Figure 3.8 shows the latency breakdown. Note that the frontend latency represents the complete end-to-end latency. First, we validate that our own implementation of DeathStarBench on Rust is a faithful re-implementation. We can see that the original Go implementation and our Rust implementation have similar latency. Moreover, the amount of latency spent in gRPC is similar. Second, mRPC with a null policy outperforms by 2.5× gRPC with a sidecar proxy in average end-to-end latency. Figure B contains more details about the tail latency and the scenario without a sidecar.

Masstree analytics. We also evaluate the performance of Masstree [164], an in-memory key-value store, over both mRPC and eRPC [112] using RDMA. We follow the exact same workload setup used in eRPC, which contains 99% I/O-bounded point GET request and 1% CPU-bounded range SCAN request. We run the Masstree server on one machine and run the client on another machine. Both the server and the client use 10 threads, with each client thread using 16 concurrent requests. The test runs for 60 seconds. The result in Table 3.3 shows that eRPC outperforms mRPC, which makes sense since eRPC is a well-designed library implementation that is focused on high performance. mRPC enables

Table 3.4: Global QoS: Performance of latency- and bandwidth-sensitive applications with and without a global QoS policy.

	Latency App		B/W App
	P95 Latency	P99 Latency	Bandwidth
w/o QoS	45.1 µs	54.6 µs	22.2 Gbps
w/ QoS	19.5 µs	21.8 µs	22.0 Gbps

many other manageability features in exchange for a slight reduction in performance. In this case, using mRPC instead of eRPC means that median latency increases by 34% and throughput reduces by 20%.

3.7.5 Benefits of Advanced Manageability Features

Next, we demonstrate the performance benefits of having centralized RPC management, through two advanced manageability features that we developed (see §3.5). We use synthetic workloads to test the advanced manageability features.

Global RPC QoS. We enable our cross-application QoS policy that reorders requests from multiple applications and prioritizes small RPC quests. We set up two applications and pin them to the same mRPC runtime. One application is latency-sensitive, sending 32-byte RPC requests with a single RPC in-flight; the other is bandwidth-sensitive, sending 32 KB requests with 64 concurrent RPCs. We measure the tail latency for the latency-sensitive application and the utilized bandwidth of the bandwidth-sensitive one.

Table 3.4 shows the result. Without the QoS policy, the bandwidth-sensitive application has a high bandwidth utilization; however, the latency-sensitive application suffers from a high tail latency. With the QoS policy in place, the small requests from the latency-sensitive application get higher priority and are sent first, improving P99 latency from 54.6 µs to 21.8 µs. Since small RPC requests consume negligible bandwidth, it barely affects the bandwidth-sensitive application (less than a 1% bandwidth drop).

RDMA Scheduler. Our RDMA scheduler batches small RPC requests into (at most) 16KB messages and sends requests using a single RMDA operation to reduce the load on

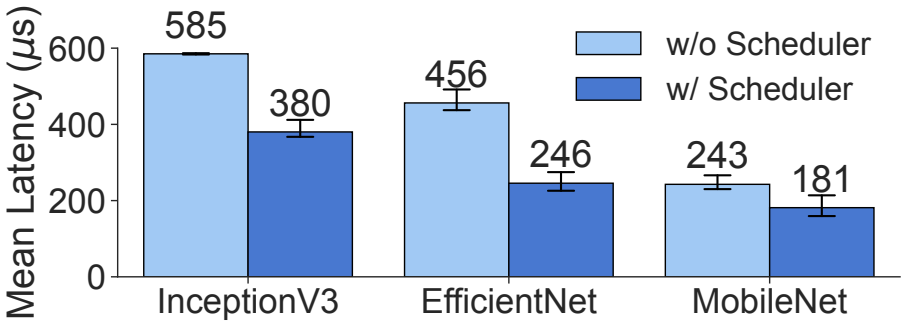


FIGURE 3.9: RDMA Scheduler: Mean RPC latency with or without RDMA scheduler. The error bars show the 95% confidence interval.

the RDMA NIC. Our synthetic workload is based on BytePS [109], which uses RDMA for distributed deep learning. To synchronize a tensor to/from a server, BytePS prepends an 8-byte key and appends a 4-byte length to describe the tensor. The three disjoint memory blocks are placed in a scatter-gather list and submitted to the NIC, resulting in a small-large-small message pattern that triggers a performance anomaly [128]. This message pattern is quite common in real applications, as programs often need to describe a large payload with a small piece of metadata. We emulate BytePS's RPC request pattern and generate RPCs from three widely-used models: MobileNet, EfficientNetB0, and InceptionV3 [85, 265, 264]. Each RPC call consists of an 8-byte key, a payload of tensor, and a 4-byte length. We use a single thread to make RPCs. Figure 3.9 shows the average RPC latency. The RDMA scheduler provides 30-90% latency improvement. This improvement differs for different neural networks, because of different RDMA message patterns.

3.8 Related Work

Integrating collective communication into the network. There are several prior efforts in integrating collective communcation into the network. ATP [133], SwitchML [244], and PANAMA [65] propose offloading AllReduce operations to in-network hardware to enable multi-tenant distributed machine learning. The key difference is that MCCS targets at the public cloud environment, where these works all require tenant applications to be trusted. In these works, a misbehaving or malicious application can circumvent the cloud provider

designed collective communication strategy, and this will require well-behaving tenants to adjust their strategies accordingly. In MCCS, all the collective operations are managed through the MCCS daemon. Another difference is that MCCS's performance gain is not from in-network gradient aggregation but from dynamic adjustment to collective communication strategy.

Exposing public cloud network information for tenants to pick collective communication strategies. A separate line of work focuses on letting tenant acquire information about the physical network of the cloud provider in order to pick collective communication strategies. NetHint [33] presents an approach that the cloud provider periodically exposes a hint, containing a subset of the physical network topology and link utilization, to help the tenant pick collective communication strategy. However, a cloud provider may have security and privacy concerns of exposing their physical network topology and network utilization to cloud tenants to prevent adoption. PLink [161] and Choreo [132] let tenant applications measure their VM-level network bandwidth in order to pick collective communication strategies or decide on job scheduling. These approaches are not guaranteed to be accurate, because reverse engineering the network configurations from a single tenant's observation is generally hard. Further, in both approaches, tenants are making their own decisions on collective communication. In comparison, MCCS controls all tenants' collective communication.

Choosing collective communication strategies based on network topology and bandwidth. Optimizing collective communication strategies for particular network topology and bandwidth configuration is a standard task for developers running large-scale workloads on supercomputers [46, 119]. For machine learning workloads, several prior works have focused on improving collective performance [208, 282, 249]. These works all focus on the single-tenant scenarios. Our work focuses on the multi-tenant public cloud setting. We need to deal with challenges of dynamically changing collective communication strategies, which is not a concern in single-tenant scenarios.

Quality of Service (QoS) in a multi-tenant network. How to let multiple tenants share a cloud network with QoS guarantees is an old topic. A cloud datacenter network often

uses a combination of congestion control [229, 4, 315, 79], load balancing [3, 121, 302, 198], and various types of rate limiting techniques [12, 106, 131, 136, 216, 5]. These works focuses on how to share bandwidth given a set of point-to-point network demand. The optimizations mRPC addresses is on having multiple collective communication operations share the bandwidth by selecting collective communication strategies (e.g., the ordering of nodes in an AllReduce ring for each tenant), which is a different and complementary problem.

3.9 Summary

Remote procedure call has become the de facto abstraction for building distributed applications in datacenters. The increasing demand for manageability makes today's RPC libraries inadequate. Inserting a sidecar proxy into the network datapath allows for manageability but slows down RPC substantially due to redundant marshalling and unmarshalling. We present mRPC, a novel architecture to implement RPC as a managed service to achieve both high performance and manageability. mRPC eliminates the redundant marshalling overhead by applying policy to RPC data before marshalling and only copying data when necessary for security. This new architecture enables live upgrade of RPC processing logic and new RPC scheduling and transport methods to improve performance. We have performed extensive evaluations through a set of micro-benchmarks and two real applications to demonstrate that mRPC enables a unique combination of high performance, policy flexibility, security, and application-level availability. Our source code is available at `https://github.com/phoenix-dataplane/phoenix`.

3.10 Acknowledgment

The mRPC project is in collaboration with Jingrong Chen. Jingrong focused on the design and implementation of the framework of mRPC service, RPC marshalling, transport engines and polices. I contributed to the design and implementation of mRPC schema compiler, memory management features and live upgrade capability.

4. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud

In Chapter 3, we show that how we rearhcitect RPCs as a managed system service, instead of using the library and sidecar based solution, to efficiently enforce policies while enabling new manageability features. Next, we move to collective communication. We find that existing library based approach to implement collective communication is ill-suited in a multi-tenant cloud environment. They miss many performance optimization opportunities due to the lack of information of both network topology and other tenants. In this chapter, we propose MCCS to address this issue, our alternative architecture where collective communication abstractions are directly provided and implemented by the cloud provider.

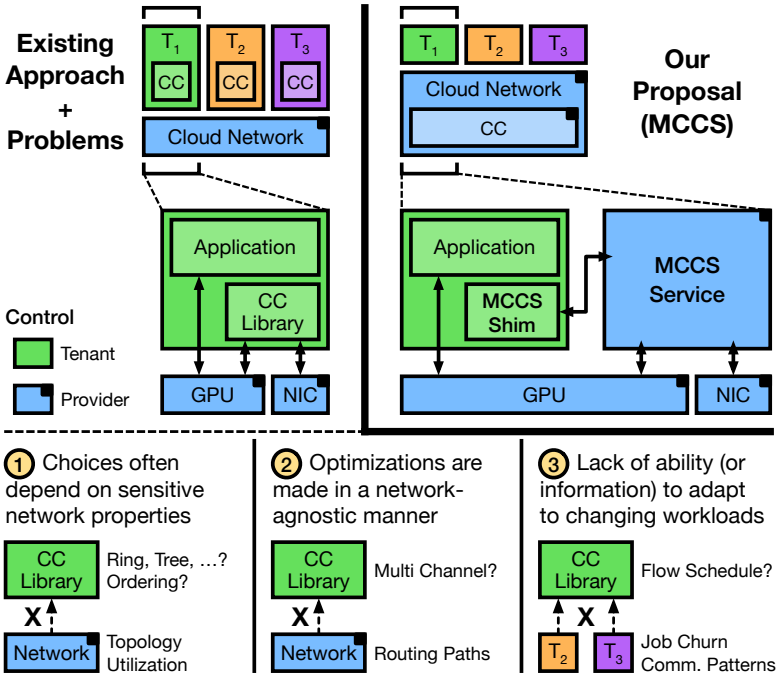


FIGURE 4.1: Comparison between existing approaches and our approach (MCCS) to collective communication.

4.1 Introduction

Today, distributed workloads have increasingly moved to the cloud for the ease of infrastructure management and resource pooling. In a public cloud environment, however, existing collective communication libraries have shown several shortcomings. First, choosing the most efficient algorithm requires knowledge of the physical network topology and link utilization, which are not available to cloud tenants. As a consequence, this may lead to sub-optimal decisions between ring- and tree-based algorithms and their configurations (e.g., participant ordering in a ring). Second, current libraries (e.g., NCCL) decide the exact strategy at initialization time and will not change the chosen strategy once the job starts. While this is fine for single-tenant settings (e.g., supercomputers), it is not ideal for multi-tenant settings because the best choice of algorithm depends on the other tenants' communication patterns. Finally, current libraries often make optimization choices in a manner that is agnostic to the underlying physical network configuration; however, these optimizations frequently rely on assumptions regarding the configuration. For instance, NCCL instantiates multiple TCP/RDMA connections between nodes to improve throughput by exploiting multiple network paths in parallel even though the connections may be routed via the same (shared) physical path.

In this chapter, we propose a new approach that more tightly integrates collective communication with the cloud network instead of the applications. We call our approach MCCS, which is short for Managed Collective Communication as a Service. MCCS exposes traditional collective communication abstractions to applications, yet decouples the implementation from the applications themselves to extend more control to the cloud infrastructure provider. With MCCS, tenants are no longer responsible for implementing collectives, which often relied on information unavailable to the tenants themselves (e.g., physical configuration, properties of other tenants' applications). Meanwhile, MCCS extends significant flexibility to the cloud provider to support a variety of benefits: First, the provider can easily adopt custom, proprietary collective communication approaches

without the need for changing existing user applications. Second, the provider can enforce fine-grained quality of service (QoS) policies at the level of collective operations. Third, the provider is no longer forced to choose between providing strong performance or the confidentiality of their proprietary infrastructure.

Achieving our goals for MCCS requires us to address several key challenges. First, we need to resolve the tension between decoupling collectives from the application while maintaining the existing interface. Second, collective communication is a group operation that requires synchronization among a number of components (e.g., application, service, hardware). Third, we need to support policies driven by the changing status of the cluster which can improve performance at both the logical-level (e.g., ring strategy) and physical-level (e.g., flow scheduling).

We implement a prototype for MCCS that targets applications using GPU-based computation and provide a lightweight shim library that connects applications with our system service. Our system can easily integrate existing collective algorithms implemented in NCCL through their CUDA kernels, as well as support more-customized algorithms. We evaluate the performance of MCCS using a small scale testbed and large scale simulations. Our testbed results have shown that MCCS consistently outperforms NCCL by up to 2.4x in terms of algorithm bandwidth, and improves training workloads in a multi-tenant improves by up to 34%. Our simulation results have demonstrated that MCCS enables an overall speed-up of 3.4x on a large-scale cluster. Our source code is available at `https://github.com/phoenix-dataplane/MCCS/`.

In this chapter, we make the following contributions:

- A new architecture for supporting collective communication in multi-tenant scenarios, shifting control from applications to the cloud network to improve performance.
- An approach to enable dynamic reconfiguration of collective implementations at runtime, which we leverage to demonstrate strong policies such as collaborative transfer scheduling across applications.
- A prototype implementation of MCCS targeting distributed machine learning work-

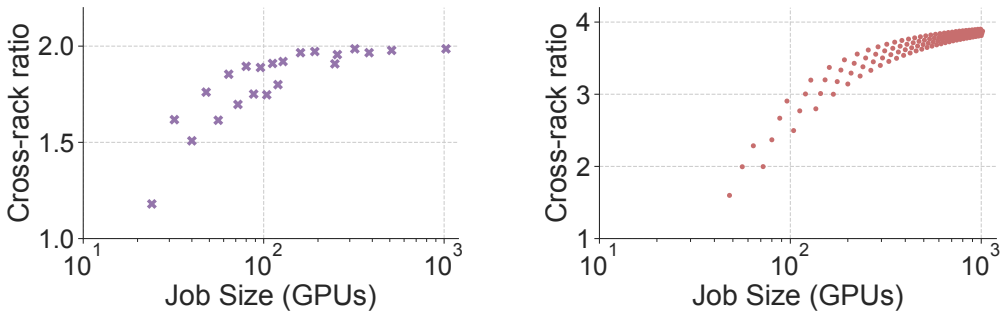
loads that is conceptually a drop-in replacement for NCCL, along with an evaluation that demonstrates the benefits via real-world traces on a testbed deployment and simulator.

4.2 Using Collective Communication Libraries in a Multi-Tenant Network?

In a multi-tenant datacenter network, traditional collective communication libraries face several challenges. Firstly, cloud networks often provide a simplified, black-box abstraction, where all tenant instances are connected through a big, virtual switch. However, these instances may actually be distributed across multiple physical racks, which are interconnected to upper-layer switches through multiple links, sometimes with oversubscription.

This lack of visibility into the physical topology can lead to suboptimal collective algorithm selection or configuration. For instance, in a ring-based collective algorithms where data transfer follows worker ranks (which are assigned by users), Without topology information, randomly assign ranks to workers in different racks could lead to the ring to cross racks back and forth multiple times, causing substantially more inter-rack traffic than necessary.

Figure 4.2 illustrates the network overhead introduced by non-optimal ring configuration with respect to job sizes. We have a production trace collected at one of the largest social network company, whose production cluster uses a spine-leaf architecture. Each rack connects two hosts, each with 8 GPUs and 8 NICs. We measure a job's network overhead using cross-rack ratio, where is the number of cross-rack flows of the collective ring used by the job, normalized to that of an optimal ring configuration. A ring configuration in the worst case introduces 2x cross-rack flows compared to the optimal one. The performance degradation would only grow as more hosts are placed under a rack. We simulate a cluster with the same scale as the company's and computes the expected cross-rack ratios with different job sizes, if ring ordering is randomly chosen and we assume jobs are perfectly packed to hosts. The worst case overhead becomes 4x in this scenario. We also find that the overhead grows with respect to the job size.



(a) [Empirical] 2 hosts/rack

(b) [Simulated] 4 hosts/rack

FIGURE 4.2: Number of cross rack flows normalized to optimal ring from both production trace and simulation.

Further, in today's multi-path datacenters, the most commonly used network load balancing strategy is Equal-Cost Multi-Path (ECMP). However, ECMP may not efficiently handle multiple flows from a ring-based collective operation, potentially leading to congestion on a single physical path and reduced throughput.

NetHint [33] suggests letting a cloud provider expose its network topology and link utilization. This transparency will potentially enable a collective communication library adjust its choices of collective communication at runtime. However, this approach raises security and privacy concerns, because a cloud provider has incentive to maintain the confidentiality of its network topology and link utilization.

In summary, cloud tenants face a significant challenge in selecting an optimal collective communication algorithm due to the lack of visibility into the physical network's structure. Providing tenants with access to this information could introduce security risks for cloud providers, as it exposes sensitive details of their infrastructure. *A viable approach appears to be for the cloud provider to assume responsibility for choosing the collective communication strategy on behalf of the tenant.* Our system, MCCS, explores this approach.

4.3 Overview

MCCS is a new design of collective communication for the multi-tenant cloud setting. We have following goals. First, a cloud provider can decide the collective communication

strategy for a cloud tenants. The cloud tenant calls a higher-level collective communication interface like AllReduce, instead of instantiating point-to-point data transfer by a collective communication library. Second, the tenant has no knowledge of what algorithm is chosen and does not know the cloud's network topology, link utilization, etc. The cloud provider hides all these sensitive information inside a cloud service. Third, the cloud provider can change the collective communication strategy without interrupting the running tenant to accommodate changes in the multi-tenant network (e.g., to accommodate a new tenant's workload, to leverage more available bandwidth as other tenants leave the network). Finally, our new design will enable the cloud provider to do other optimizations (e.g., joint optimization of collective communication and flow scheduling, having multiple tenants' workloads use the bandwidth in an interleaved fashion).

MCCS, realizes collective communication as a cloud service instead of an application library. At the same time, MCCS maintains a similar interface as traditional collective communication libraries such as NCCL. Figure 4.1 presents an overview of the various components of MCCS and its architecture different compared to traditional collective communication libraries. MCCS service runs as a trusted, user-space process with access to all GPUs and NICs on the host. User applications on the host only have access to one or more GPUs allocated to the application. Applications are compiled with MCCS shim library, which communicates with MCCS service using shared host and GPU memory. MCCS service is controlled by the cloud provider, and applications can only access it through its collective communication APIs.

MCCS service issues collective communication operations for GPU buffers on behalf of the cloud tenant. MCCS service decides the collective communication strategy and can change the strategy at runtime. Because MCCS service is controlled by a cloud provider, for every flow in the collective communication, MCCS service can also decide its network path in the cloud network using source routing or other path control schemes. Further, MCCS service can enforce QoS policies by controlling flow paths and the timing of collective communication operation execution.

4.4 Design

In this section, we break down and discuss the design of MCCS in three parts. First, we look at how MCCS meets the existing collective API while decoupling collectives from the application. Next, we discuss how MCCS supports the data path for collective communication with support for runtime reconfigurability. Finally, we present our solution to decouple policy from mechanism and enable flexible management of collectives. While our design is agnostic to any particular collective communication implementation, we focus on the NCCL throughout this section; other GPU-based collective communication libraries such as RCCL [234] and oneCCL [199] maintain similar semantics and terminologies as NCCL.

4.4.1 Collective Interface

To support collective communication as a service, MCCS needs to: 1) provide an interface to applications for invoking collectives, and 2) enable synchronization between application computation and collective operations. One goal that constrains our design is a desire to maintain as close to the same interface as existing libraries, like NCCL, so that MCCS could act as a drop-in replacement.

NCCL provides APIs that build on core CUDA primitives for applications to issue collectives. A CUDA stream is similar to the notion of a thread, allowing applications to enqueue a sequence of operations (e.g., kernels) to be executed in-order by the GPU. An application can use multiple threads to express concurrency between operations by enqueuing them on different streams. When invoking a NCCL collective API, developers specify the stream that the collective will be enqueued on; this is used to capture the data dependency between a collective and the (prior) computations that generate the data for the collective to operate over.

To realize our design for MCCS, we need to address two challenges that relate to the inherent isolation between applications that leverage CUDA. First, similar to host memory, GPU memory between different processes is isolated by default. How can MCCS service

GPU buffers of applications and implement collectives on them? Second, due to the fact that CUDA streams are limited within a single process, how can MCCS service's APIs maintain the same thread semantics for the application's CUDA calls, as respected by NCCL?

We will discuss each of these in turn.

Memory Management. We address the memory management challenge by choosing to redirect control over GPU memory allocations and deallocations to the MCCS service. Our shim library provides APIs that applications can invoke directly to allocate memory that will be accessible to both the application and the MCCS service; the application can use the existing CUDA APIs to manage private buffers that are not directly used as part of collective operations. Alternatively, to minimize the changes to existing applications, we also support the redirection of all allocations to the MCCS service.

The MCCS shim issues an allocation request to the MCCS service over the shared memory command queue between the application and the service. A dedicated front-end engine for the given application will handle the request by internally allocating memory on the specified GPU device and obtain an inter-process memory handle to share with the application. The MCCS shim receives and opens this handle to obtain the underlying device pointer to the allocated memory, which it returns to the application. The application can then use these pointers freely for invoking compute operations, while for collective operations, MCCS shim passes an identifier for the memory allocation and an offset to MCCS service. The service will check whether the data buffer user passes is within a valid allocation before performing the operation. This process follows similarly for deallocation requests – the shim is responsible for closing the inter-process memory handle before forwarding the request to the MCCS service.

Synchronization. We address the synchronization challenge by designing an event-based mechanism that maintains the traditional semantics of CUDA streams, which enables the sequential ordering of dependent compute and collective operations. Since CUDA streams belong to a single application and cannot be shared (unlike GPU memory via inter-process

handles), we need an alternative approach for MCCS. We leverage the CUDA event primitive, which provides the ability to enqueue a stream operation that blocks waiting on the notification of a event that was enqueued on a (different) stream. Unlike streams, events can be shared via inter-process handles.

MCCS addresses synchronization at the level of communicators, a standard abstraction used in collective communication libraries (e.g., NCCL) to specify a subset of n nodes, each assigned a unique rank in $[0, n)$, to take part in collective operations. While an application may use an arbitrary set of streams for managing its computation and collective operations, the MCCS service maintains one stream per communicator. When an application issues a collective operation for a given communicator, the MCCS service enqueues the communication kernels that implement the collective request on the associated stream.

To enable communication kernels on MCCS-managed streams to wait for compute kernels on application-managed streams to finish (and vice-versa), we need to introduce event management operations into the MCCS shim. When a communicator is created, the MCCS service also creates a corresponding event object and obtains an inter-process event handle that it returns to the MCCS shim (along with the communicator handle). The MCCS shim will use this event to enqueue an operation on the application's stream to block waiting for the collective to finish prior to executing any subsequent operations. Likewise, the MCCS shim also creates an event object for each application stream to share with the MCCS service. Instead of hooking directly into CUDA's stream management API, the MCCS shim creates events in an on-demand fashion whenever a new application stream is used for invoking a collective operation. The MCCS shim shares an inter-process event handle along with a stream identifier with the MCCS service, which it uses to enqueue an operation on the internal stream for the communicator to block waiting for any computation to finish prior to executing the subsequent communication kernel.

Now, we have developed an approach for decoupling the collective communication from the application while still providing the same interface from our MCCS service. Next, we will explore how MCCS actually implements the communication that underlies collective

operations while enabling more manageability as compared to existing approaches.

4.4.2 Collective Communication

To support dynamic reconfiguration of the data path subject to policies, we need to decouple the data path setup from the communicator's initialization. In NCCL, most of the work that configures the data path takes place when a communicator is created by an application, where NCCL first attempts to detect the intra-host topology (e.g., NVLink between GPUs) to figure out how to connect all of the intra-host GPUs, while also identifying the best NIC to use. After the intra-host topology is decided, each rank communicates with the root (i.e., rank 0) to form an AllGather TCP/IP-based ring for exchanging control information. At this point, NCCL can set up the underlying algorithm for implementing collective operations by constructing rings and trees, which it uses to establish peer-to-peer connections between nodes (e.g., consecutive ranks in the ring).

NCCL is mainly designed to run on infrastructure that is tightly controlled by the user (i.e., application developer), focusing only on optimizing the intra-host strategy while leaving the optimization of inter-host strategy to users. In particular, NCCL simply connects inter-host rings and trees according to the ordering of user-specified ranks when establishing the communicator; therefore, users must carefully design the GPU-to-rank mapping, which requires expert knowledge of the cluster topology (and is often error prone). In multi-tenant settings for public cloud, where the users and infrastructure provider are not the same principal, this becomes problematic. However, even intra-host optimization presents issues in public cloud settings as well due to virtualization. NCCL can potentially fail to optimize intra-host strategy because it relies on `sysfs` information to discover the PCIe topology of GPUs and NICs. Typically, modern virtualization approaches may hide such information from tenants (and thus from tenant-controlled collective libraries like NCCL). This challenge has been noted in other recent work on collective communication algorithms such as TACCL [249].

By decoupling the implementation of collectives from the applications, we are uniquely

positioned to transform these prior challenges into opportunities for MCCS. First, we can leverage proprietary (and thus often confidential) topology information within the context of the MCCS service without revealing such information to the applications. This involves architectural challenges in terms of running collective communication strategies outside the tenant application's control and observability. Second, we can enable dynamic reconfiguration of collective strategies in response to changes in the cloud network (or the set of multi-tenant applications). This involves addressing a key challenge for enabling reconfiguration (e.g., at the level of ring orderings) that simultaneously achieves high-performance and ensures synchronization across the participating nodes within a communicator. We will discuss each of these in turn.

Multi-Tenant and Topology-Aware Architecture. For MCCS, we need to develop a new architecture to simultaneously support multiple applications sharing cluster, or even individual host, resources while also being able to exploit low-level network information (e.g., physical topology). Given that NCCL is focused on a single application, the implementation of collective communication for communicators consists of a "transport agent", which is responsible for managing the sending/receiving of inter-host collective communication traffic via available NICs based on GPU data buffers. We choose to decompose the role of the MCCS service into two main engines [1]: 1) a proxy engine, and 2) a transport engine.

The proxy engine is responsible for bridging the gap between high-level communicators and low-level resources. For each GPU on a given host, MCCS initializes a single proxy engine that handles all communicators which include that GPU in their ranks. When a collective is issued, the proxy engine manages the higher-level collective strategies and network configurations for how the collective communication will be implemented. For instance, this enables MCCS to optimize how inter- and intra-host rings are connected and ordered for improve resource utilization. Additionally, MCCS enables the incorporation of various collective strategies optimized for specific topologies, such as those proposed

[1] We use the term "engine" to refer to a general wrapper around functions that can asynchronously operate over inputs to generate some outputs.

in recent research [249, 27, 203] or even proprietary strategies developed in-house by the provider. In all cases where communication takes place over intra-host communication channels (e.g., NVLink, host shared memory buffer), the proxy engine manages the setup and use of those channels directly.

For all inter-host communication, the proxy engine offloads the management to the transport engine. While conceptually similar to the transport agent in NCCL, the transport engine in the MCCS service is responsible for multiple applications simultaneously. Additionally, the transport engine is responsible for providing the underlying mechanisms for scheduling flows on network paths using existing path control techniques (e.g., source routing, policy-based routing). There may be one or more transport engines associated with each GPU to support more communication parallelism.

Dynamic Reconfiguration. The MCCS service exposes support for dynamic reconfiguration via a command that is made available to the provider (not the applications). A key goal of our design is to ensure that the performance overhead for performing a reconfiguration is low (since this otherwise reduces the benefit from implementing smart policies) and that there is zero (or negligible) performance overhead for collective operations when no reconfiguration is issued. At a high-level, this motivates our choice to support reconfiguration at the granularity of collective operations. Reconfiguration should be a coarse-grained scheduling decision in practice, reacting to events such as link utilization increasing due to traffic that is outside the scope of collectives managed by MCCS (e.g., fetching training data, background flows).

While it is straightforward for all nodes to agree on a configuration at initialization time, which necessarily takes place before collectives, this is much more challenging when implementing reconfigurations between collective operations. For an illustration of this, consider the example shown in Figure 4.3. Here we assume an application created a communicator consisting of three GPUs and that it issues a series of AllReduce (AR) collectives for that communicator. We differentiate between the launch of a collective, which shows the ring configuration, and the subsequent completion of a collective. At some point, a reconfigura-

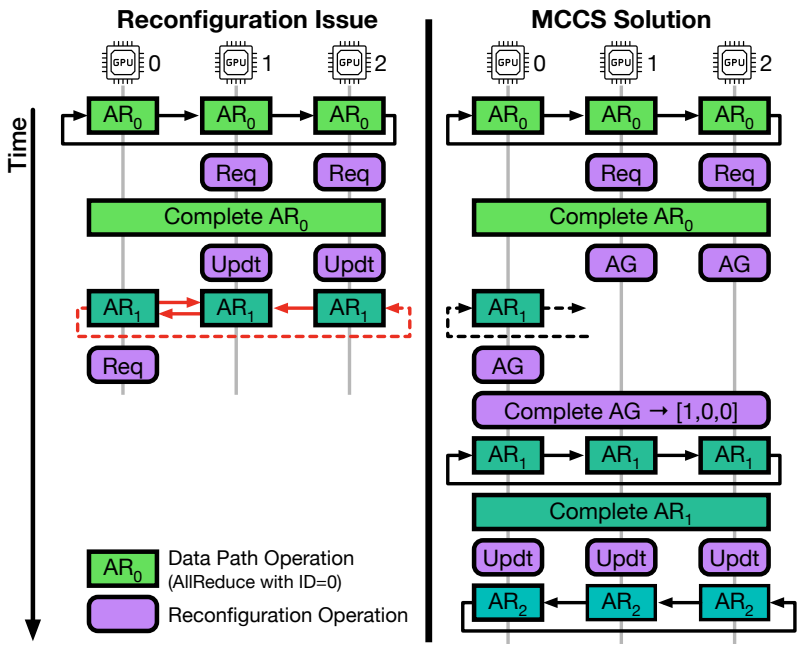


FIGURE 4.3: Example showcasing a potential synchronization issue in handling dynamic reconfigurations (left) and the MCCS protocol to address this (right).

tion request (Req) is sent to each of the MCCS service instances running on different nodes; however, due to arbitrary network and processing delays, it is possible for the command to be received and processed at different times. Without appropriate synchronization, this could lead to correctness issues (shown on the left) in which rank 0 executes AR_1 from the perspective of the previous ring ordering, while ranks 1 and 2 perform updates (Updt) to handle the reconfiguration request prior to AR_1. We need to address this problem without requiring expensive synchronization operations on the fast path (i.e., between collectives when no reconfiguration takes place).

Our solution is to leverage the per-communicator control ring as the basis to construct an efficient barrier synchronization mechanism. Each proxy maintains a sequence number for the collectives over time, which inherently matches across all nodes in a communicator because each collective involves every node. After receiving a reconfiguration request, each proxy enqueues all subsequent collectives prior to issuing an AllGather (AG) collective on the control ring to exchange the sequence number corresponding to the last collective

launched. Local updates will not be completed until the AllGather completes, which will provide all nodes with the sequence numbers for every other node; computing the maximum sequence number enables nodes to identify which collectives should precede any reconfiguration update (i.e., collective sequence number is less than or equal to the maximum).

Looking back at Figure 4.3, we see on the right how this synchronization prevents this correctness issue in this example. When the proxies for ranks 1 and 2 receive the reconfiguration request (Req), they issue the AllGather operation with their data containing 0 for the latest collective (AR_0) that was launched. Later, when the proxy for rank 0 receives the reconfiguration request, it also issues the AllGather operation; however, since it already launched AR_1, its data contains 1 for the latest collective. At this point, the AllGather operation completes, which allows the proxies for all ranks to determine that the maximum sequence number is 1; for ranks 2 and 3, this means that they should issue the queued AR_1 collective prior to updating the configuration. The updated ring ordering will be used in all future collectives until a another future reconfiguration request is issued. To update a configuration, the proxy engines will interact with the transport engines to close all existing peer-to-peer connections for the communicator and clean up corresponding resources. Afterwards, the new connections are instantiated based on the chosen strategy (e.g., rank ordering within a ring), similar to what is performed at the time of initialization.

In analyzing the performance implications of this design, we can make two observations. First, issuing a reconfiguration request can introduce some performance overhead, since collectives will be stalled until the AllGather for the reconfiguration is complete (i.e., until the last proxy receives and handles the request). Additionally, there is some overhead in tearing down and establishing new peer-to-peer connections. As we will demonstrate in the evaluation, the performance overhead for handling reconfigurations is rather small, and enables significant performance benefits from smart policies. Second, in the absence of a reconfiguration request, there is no performance overhead. Note that the proxy for rank 0 is able to launch the AR_1 collective before even being aware that the other ranks received a reconfiguration request – any synchronization via the control channel (or blocking) only

occurs after a request is received.

At this point, we have the ability for applications to issue collectives and for the MCCS service to implement them while supporting reconfigurations at runtime while taking into account low-level topology information. Next, we will explore how MCCS enables flexible and expressive management according to provider-defined policies that build on top of the reconfiguration mechanisms that we just discussed.

4.4.3 Enabling Manageability

One of our key goals in MCCS is to cleanly decouple policy from mechanism. The design of our proxy and transport engines within the MCCS service enables management of both the control and data paths for collective communication. On the control path, the MCCS service can support different collective strategies for various applications as well as control network resource allocation (e.g., NICs per application, network routing). On the data path, the MCCS service can support fine-grained control of communication through augmentation of the transport engine to control the conditions for sending network traffic. Our architecture enables this through dynamic loading of provider-supplied logic that can handle policy decisions determined by an external controller.

To enable an external controller (e.g., centralized manager) to schedule the collective communication across all applications on the cluster, the MCCS service needs to provide an interface for exposing necessary information. For each application, this information is based on the set of active communicators, including the set of GPUs (and hosts) that make up the ranks within the communicator, and the current configuration of collective strategy (e.g., ring configuration) and network resources (e.g., flow mapping). Additionally, the MCCS service can perform fine-grained tracing of collectives issued by applications to determine properties of their computation and communication patterns. The controller consumes this data to make a policy decision.

Next, we look at several concrete examples of scheduling and quality-of-service (QoS) policies, which will also be used in our evaluation of MCCS. While these examples are admit-

tedly straightforward, they effectively illustrate MCCS's system capabilities beyond what today's collective communication library can offer. MCCS can also incorporate topology-optimized collective algorithms from MSCCL [249, 27], while they only apply to a single-tenant environment.

Topology-aware collaborative scheduling. We explore the following two heuristics to enable the joint optimization of the algorithmic strategy at the collective level and flow assignment at the network layer.

Example #1: Locality-aware ring configuration. The ordering of how the hosts are chained in the ring collective algorithm directly dictates the overall communication pattern. If many flows have to go through links above the leaf level (assuming a Clos network topology), severe congestion could occur due to over-subscription. Hence, our goal is to minimize the number of cross-rack / cross-pod flows. We apply a greedy algorithm to configure the ring ordering for each communicator (application). We group the participant hosts by their locality (e.g., under the same rack, under the same pod) and then connect them in a sequential order. The algorithm takes the set of participant GPUs for each communicator obtained by MCCS service management APIs, and sends an optimized ring ordering back to MCCS service.

Example #2: Best-fit fair flow assignment (FFA). Once the ring configuration for all applications are optimized, the communication patterns between hosts and hence the set of flows can be determined. Still, using the standard ECMP approach to map flows into network routes could lead to significant overall collective performance degradation and inconsistency due to flow collision. Our goal is to maximize the aggregated collective performance of all applications, and ensure fairness between different applications. We use a slightly modified version of the greedy heuristics proposed in Hedera [53], where for each flow we assign it the path that has minimal excess bandwidth demand. We round-robin between flows from different jobs for fairness. For example, if two applications are both performing collectives using hosts on rack A and B. There are 2 routes between A and B and each application have 2 flows from A to B. FFA would assign each route a flow from

both application. FFA takes the collective strategy configuration of all communicators as input. As communication patterns solely depend on the collective strategy, FFA knows all flows (RDMA connections) in the network. It then assigns each flow a route ID, where the mapping is issued to MCCS service.

QoS features. MCCS enables priority control at both coarse-grained resource allocation and fine-grained communication.

Example #3: Priority flow assignment (PFA). We modify FFA to allow some routes to be reserved for high priority applications. We first fit flows of low priority applications using only non-reserved routes, and flows of high priority applications are assigned best routes from all available ones. In our example, PFA can dedicate one of the two routes between rack A and B to the prioritized application.

Example #4: Traffic scheduling (TS). With priority flow assignment, we can dedicate networks links to some of the highest priority applications. However, we may still have some applications sharing links. MCCS could enforce a traffic schedule to control when each application can send out traffic. In our implementation, we apply a simple time window based approach inspired by CASSINI [230] to interleave traffic. TS invokes MCCS tracing API and requests a trace of a prioritized application. TS then analyzes the idle cycles of the application when it is not issuing collectives. TS sends a time interval schedule to MCCS service. Transport engines in MCCS service then allow other applications to send traffic only when the prioritized application is idle.

4.5 Implementation

MCCS is implemented in 13.5K lines of Rust: 1.5K for the shim library and IPC implementation, 6K for control and management planes, and 6K for transport engine and transport protocols.

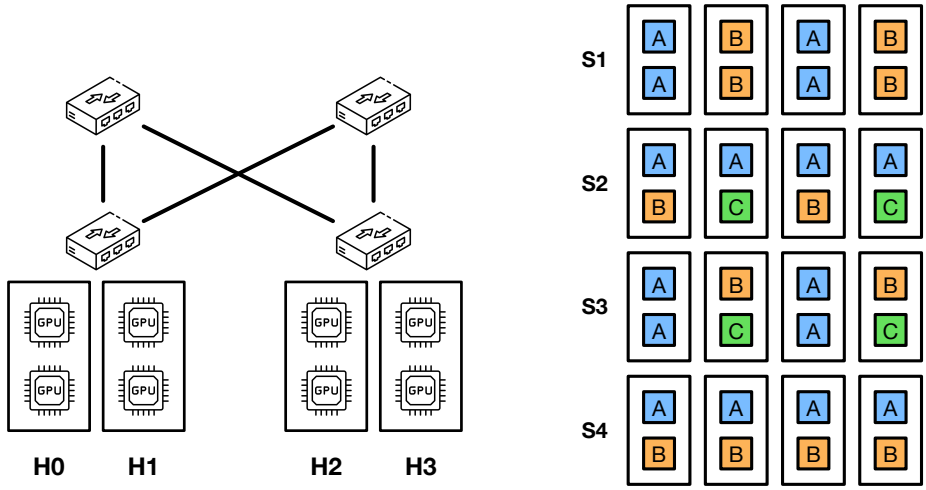
Collective CUDA kernels and transports. We adapt CUDA kernels from NCCL v2.17.1 for computation and intra-host communication. We modify the kernels so that the communicator resources on the kernel side (e.g., ring buffers) can be set up by proxy engines in the

MCCS service. We focus on ports of NCCL's ring AllReduce and AllGather kernels; however, it is straightforward to implement other collective operations, P2P communication, and other algorithms (e.g., tree algorithms). For transport protocols, we implement support for channels using host shared memory and RDMA; other channels, such as NVLink, can also be integrated.

Internal engine scheduling. Our engines are designed similar to asynchronous futures in Rust. A pool of runtimes is used to execute the engines, where each runtime corresponds to a kernel thread. Engines can be scheduled on either a dedicated runtime or a shared one. Runtimes without active engines can sleep to release the CPU. Currently, we dedicate a runtime to each engine. Compared with NCCL, which only uses an additional thread per GPU for the transport agent, our prototype would use 2 more threads for the frontend and proxy engines. However, we note that if multiple applications use the same GPU, they will share a proxy engine. We do not focus on CPU usage optimization as a core goal in our prototype, and we could implement better engine scheduling strategies to lower CPU utilization (e.g., frontend and proxy sharing a runtime if only one application uses the GPU).

Management. We leverage policy-based routing at the switch to achieve explicit route control for implementing FFA and PFA. Based on the assigned route ID for each RDMA connection, MCCS service modifies the UDP source port of ROCEv2 packets. The source port is not used by ROCEv2 protocol, hence we install a routing policy on switch that maps flows to routes based on the UDP source port specified by MCCS service. To implement TS, we currently use a hard-coded logic directly embedded in the transport engines, and we manually profile applications offline. We note that such TS scheduling logic could be easily integrated into a dynamic library function loaded by the transport engine, while the communication trace of applications can be retrieved from the MCCS management API.

For our evaluation, we consider the case in which all tenants utilize MCCS for collective communication. However, this is not strictly required. Even if only a subset of tenants use MCCS, MCCS can still collaboratively schedule the collectives of that subset, while



(a) Testbed topology (b) Multi apps setups

FIGURE 4.4: Testbed topology and multiple applications evaluation setups.

treating other flows as background flows (and adapt to them).

4.6 Evaluation

We evaluate the capabilities of MCCS using a small-scale testbed. We also conduct large-scale simulations to quantify the performance benefits of collaborative scheduling enabled by MCCS for large compute clusters.

4.6.1 Testbed Setup and Workloads

Figure 4.4a presents the setup of our testbed. We have four nodes in our testbed, each equipped with 2 NVIDIA RTX 3090 GPUs and a 100 Gbps Mellanox ConnectX-5 NIC. Using a single 100 Gbps Mellanox SN2100 switch, we emulate a spine-leaf topology with 2 leaf switches and 2 spine switches through self-wiring. Four nodes are placed under two racks, where each rack corresponds to a leaf switch. The links between the switches are limited to 50 Gbps, while the links between each host and the leaf switches are limited to 100 Gbps. This means that the over-subscription ratio of our testbed is 2. On each host, we use IB traffic class (TC) and rate limit each TC to emulate two 50 Gbps virtual NICs (one per GPU).

We use AllReduce and AllGather benchmarks to evaluate how MCCS can improve the collective performance in both the case of a single application and the case of serving multiple applications at the same time. For the single-application scenario, we use two setups: a 4-GPU setup where one GPU and one 50 Gbps NIC on each host is used, and an 8-GPU setup where all two GPUs and two 50 Gbps NICs are used. To show the effectiveness of MCCS in a multi-tenant environment, we construct 4 setups on our testbed. These setups include applications with different sizes and different placements, as shown in Figure 4.4b.

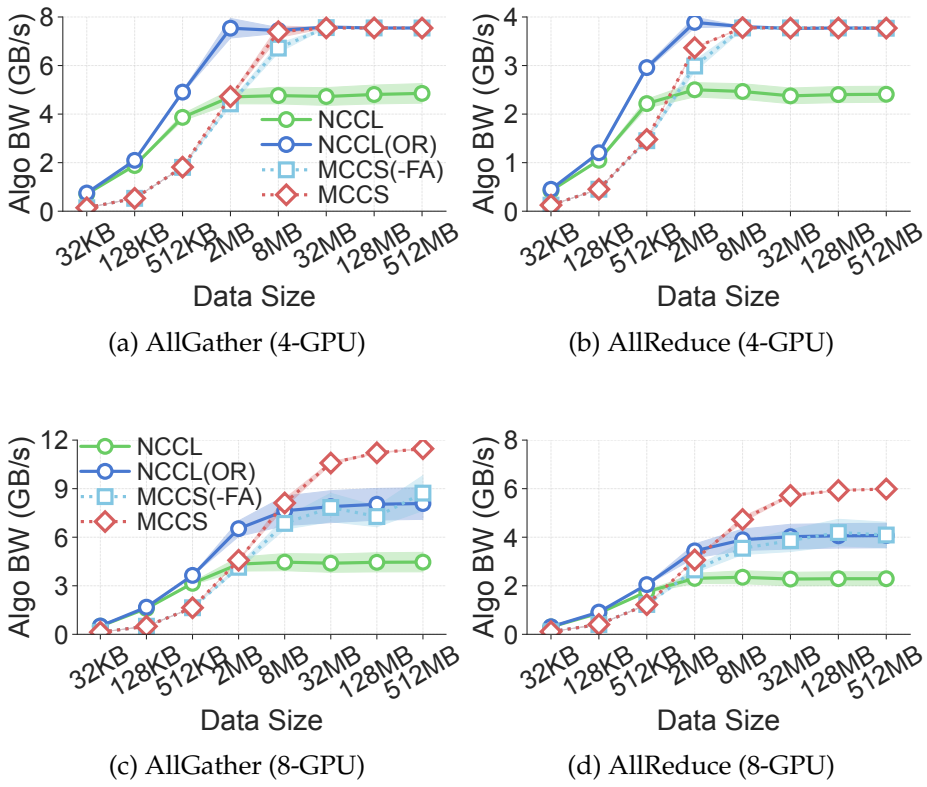
In addition to AllReduce and AllGather benchmarks, we evaluate training workloads using a traffic generator with profile traces. The traffic generator is implemented with Rust using the MCCS library. To collect the traces, we used PyTorch [207] v2.1.0, DeepSpeed [233] v0.10.3 and Megatron-LM [252] to profile a VGG-19 model [254] with data parallel training, and a 2.7B parameters GPT model [26] with tensor parallel training.

Baselines: We compare MCCS with NCCL (v2.17.1), which is not network topology aware and cannot perform inter-host ring optimization. To quantify the performance overhead of MCCS, we manually configure the inter-host ring used by NCCL with the results from our locality-aware ring configuration algorithm to serve as one of the baselines. We denote this baseline as NCCL(OR), i.e., NCCL with optimal ring.

4.6.2 Improving Single Application

We first evaluate how the performance of a single application can be improved with topology-aware scheduling capability enabled by MCCS. We run AllReduce and AllGather benchmarks of different data sizes (measured by output buffers). We report the algorithm bandwidth [193] measurement, which is calculated as output buffer size divided by execution time. To evaluate the system overhead introduced by MCCS, we also evaluate MCCS without our flow assignment algorithm, and instead rely on ECMP for routing. Figure 4.5 shows the results in the 4-GPU and 8-GPU setups. Our full solution is denoted as MCCS. We also evaluate a version of MCCS without doing flow assignment: MCCS(-FA).

MCCS's system-level performance overheads can be calculated by comparing MCCS(-



(a) AllGather (4-GPU)

(b) AllReduce (4-GPU)

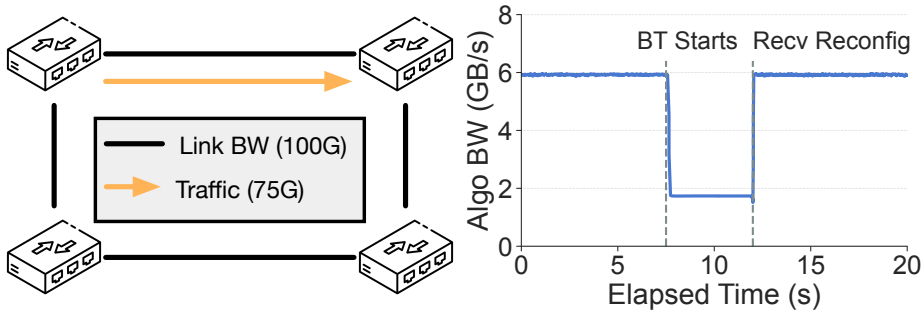
(c) AllGather (8-GPU)

(d) AllReduce (8-GPU)

FIGURE 4.5: [Single application]: Algorithm bandwidth of AllReduce and AllGather. The shaded areas represent 95% percentile intervals.

FA) and NCCL(OR), which both use the optimal ring from our ring configuration algorithm. MCCS has negligible system-level performance overheads when data size is above 8 MB. On 4 GPUs, MCCS(-FA)'s algorithm bandwidth is 63% lower than NCCL(OR) on 512 KB AllGather (which corresponds to 128 KB input per GPU) and 51% lower on 512 KB AllReduce, but the performance difference decreases to 9.7% for 8 MB AllGather and 0.75% for 8 MB AllReduce. The reason is that for large messages, MCCS's performance is bottlenecked by the collective communication's data transfer. For small messages (less than 8 MB), MCCS suffers from performance penalties due to the latency overhead introduced on the datapath. The communication between the application and the MCCS service, as well as between the internal engines of the MCCS service, incurs an overall latency of 50-80 us.

NCCL's performance is the worst because NCCL itself does not know the best ring configuration. Comparing NCCL and NCCL(OR), we find collective algorithm optimizations



(a) Example scenario (b) Ring reconfiguration

FIGURE 4.6: [Single application]: Showcase of adapting to background flows.

play a crucial role in achieving high performance. NCCL(OR) is 56% better than NCCL on the 4-GPU setup and 78% better on 8-GPU the setup for 512 MB AllReduce.

To understand how ECMP plays a role in AllReduce and All Gather performance, let's look at the 8-GPU case in Figure 4.5 because the 8-GPU scenario has cross-rack traffic. For 512 M AllReduce, MCCS outperforms MCCS(-FA) and NCCL(OR) by 46%. Note that all three approaches use optimal ring configurations. The key reason is that flow assignment is fundamental to avoiding flow collision in ECMP, so only optimizing the collective algorithm insufficient.

MCCS enables joint optimization of collective communication algorithm and flow scheduling by a cloud provider. Combining both ring configuration and flow scheduling techniques, MCCS delivers an 1.6x speed-up on the 4-GPU setup and a 2.4x speed-up on the 8-GPU setup on average for 8 MB-512 MB AllReduce and AllGather compared to NCCL.

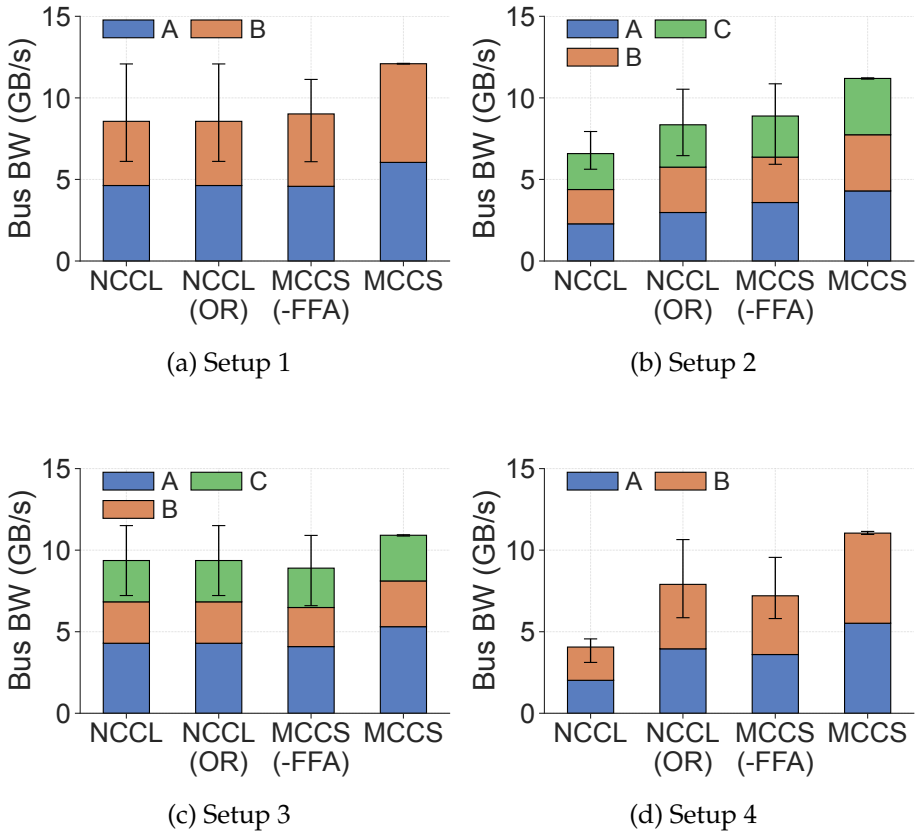
Dynamic changes of collective communication strategies to adapt to background flows. Here we also showcase the capability to reconfigure an application's collective strategy at runtime without interrupting the application. We use an example scenario to demonstrate this feature. We leave the monitoring of background flows to external components. For instance, a switch agent can be configured to report to a centralized manager when there are persistent large flows that are not managed by MCCS. The centralized manager can then send a new configuration to MCCS service. With our testbed, we emulate a topology

shown in Figure 4.6a, where each of the server is connected to a switch, and the four switches are linked as a ring. We instantiate an 8-GPU AllReduce job, the the AllReduce job uses a ring algorithm that connects hosts clockwise. As shown in Figure 4.6b, at time 7.5 s, a background flow of 75 Gbps between two switches in the clockwise direction, the available capacity for the AllReduce job drops to 25 Gbps. However, the switch links counterclockwise is not affected. If the collective strategy configuration is not adjusted, the AllReduce algorithm bandwidth drops from 5.9 GB/s to 1.7 GB/s. MCCS enables the application to recover its collective performance by transparently reverse the ring when the background flow starts. After reconfiguration command is issued (at time 12 s) by the external centralized manager, the AllReduce bandwidth immediately recovers to 5.9 GB/s.

4.6.3 Improving Multiple Applications

Next, we evaluate how MCCS improves the overall performance with a centralized view of all applications and collaborative schedule their collective communication. Figure 4.7 shows 128 MB AllReduce performance in the 4 different setups, as described in Section 4.6.1. We report bus bandwidth [193] of each application, which is normalized version of the algorithm bandwidth. Here we use bus bandwidth because it is independent of collective algorithm and the number of participants. It reflects the hardware peak bandwidth for inter GPU communication. The aggregated bus bandwidth of all applications indicates the overall network utilization, while the proportion each application gets allocated reflects fairness of allocation. For ablation study, we also compare with the baseline of MCCS without fair flow assignment. We denote this baseline as MCCS(-FFA).

For all setups, MCCS (with FFA) not only achieves the highest aggregated bus bandwidth but also ensures fairness across applications. It outperforms NCCL by 75% on average. All applications in setups 1, 2, and 4 use the same amount of NICs per host, so they should have identical inter-host GPU communication performance. MCCS therefore equally distributes the bandwidth between different applications. In setup 3, application A uses 2 GPUs and 2 NICs per host, while B and C use only 1 per host. Therefore, applica-



(a) Setup 1

(b) Setup 2

(c) Setup 3

(d) Setup 4

FIGURE 4.7: [Multi applications]: Application bus bandwidth. Error bars represent 95% percentile intervals.

tion A's inter-host collective performance should be 2 times that of applications B and C. Again, MCCS achieves fair allocation as the bus bandwidth distribution among A, B and C is close to 2:1:1. Using ECMP fails to guarantee fairness among applications. For instance, in setup 3, the performance ratio between applications A and B for MCCS(-FFA) is 1.7:1 instead of 2:1.

4.6.4 Training Workloads with QoS

We evaluate MCCS using GPT and VGG training traces. We use setup 3 for our evaluation. We assume A, B, C represent three tenants sharing the cluster. A is assigned 4 GPUs to train a VGG model from scratch on a large dataset, while B and C are assigned 2 GPUs each to finetune GPT models.

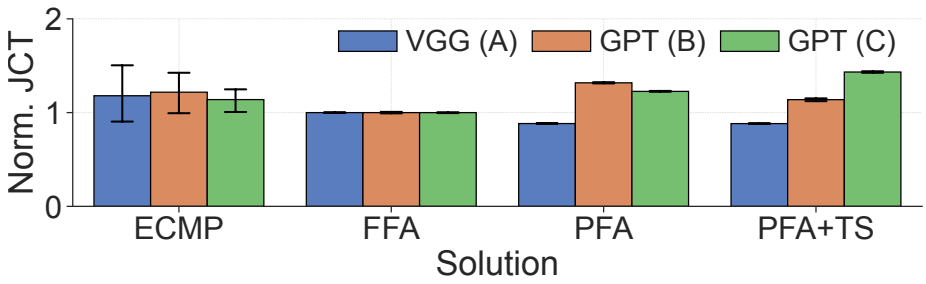


FIGURE 4.8: [Training workloads]: Job completion time using different scheduling and QoS strategies.

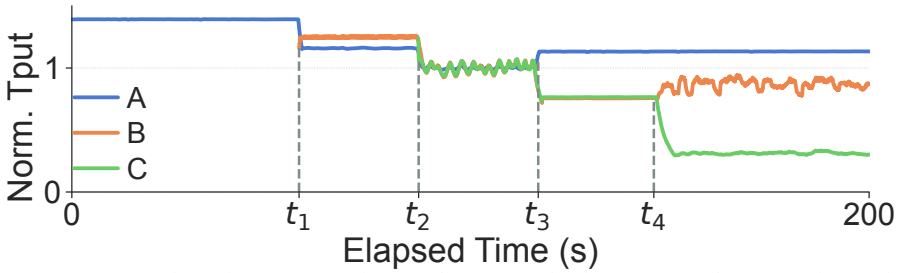


FIGURE 4.9: Normalized training throughput with dynamic job arrivals and QoS.

Fair scheduling speed-ups every workload. Using our traffic generator on MCCS to simulate the workloads, we report their job completion time (JCT) in Figure 4.8 under different scheduling approaches. Here A has the highest priority, followed by B, while C is the lowest. Error bars represent 95% percentile intervals. The JCT of each workload is normalized to its respective value under fair flow assignment (FFA). We find that ECMP routing degrades every workload. Besides having high performance variance across 10 trails, it also leads to 18%, 22%, 14% slower job completion on average, for A, B, C respectively.

QoS capabilities enable workload prioritization. Running the workloads from all three tenants at the same time inevitably result in contention of network resources. Even with fair flow scheduling, the performance of a workload would still degrade, compared to dedicate the entire network for that workload by running it independently. In this case, the infrastructure administer may prefer prioritizing some tenants. We showcase our two QoS techniques in Section 4.4.3 to demonstrate MCCS's capabilities for enabling QoS through controlling both coarse-grained resource allocation and fine-grained communication.

We assume an administrator wants to prioritize A over both B and C. Using priority flow assignment (PFA), we dedicate one of the two routes between the two racks to A, with B and C sharing the other one. PFA speeds up A's training by 13% compared to FFA and 34% compared to ECMP.

With A prioritized using PFA, B and C now shares a single bottlenecked route, so their performance degrades. If the administer wants to further prioritize B over C without affecting A, flow assignment no longer works no remaining routes are available that we can dedicate B to. Fine-grained communication-level QoS mechanism needs to be utilized. Hence, in this scenario, we apply our time window based traffic scheduling (PFA+TS) to prioritize B. Compared with PFA, in PFA+TS tenant B's training is sped up by 16%.

Dynamic policy enforcement. We demonstrate MCCS's flexibility in policy enforcement, by showing how network administers can adapt their QoS policies based on current cluster status with dynamic application arrivals. We illustrate the training throughput of A, B, C in Figure 4.9, where they arrive sequentially. The throughput is normalized to their values under FFA. A already occupies the cluster at the start, which is followed by B's arrival at t_1. As A has two 50G NICs per host, it can utilize all the 100G switching capacity of the network when there are no other tenants share the network. After B arrives, A's throughput is decreased by 17%. Then, C arrives at t_2, and all three of them share the network using FFA. The throughput of A now drops further by 14%. There are also some fluctuations in the throughput of all applications, which could attribute to network congestion. After t_3, the administer prioritizes A over B and C using PFA, A's performance therefore improves by 13%. At time t_4, the administer the further prioritizes B over C using TS, the throughput of B is increased by 18%. The fluctuations after t_3 is introduced by our time window based TS.

4.6.5 Simulations

We evaluate how a larger scale deployment can benefit from MCCS via simulations. We compare among three solutions (1) random ring selection, (2) optimal ring (OR) selection,

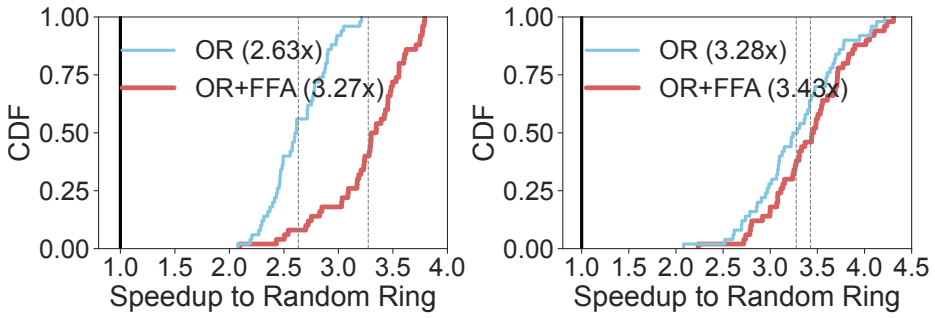


FIGURE 4.10: [Simulations]: MCCS's speedup of AllReduce completion time compared with random ring.

and (3), OR with fair flow assignment (FFA). In OR, we always create optimal rings, with the number of rings equal to the number of network multi-path choices. In OR+FFA (representing MCCS), we assign each ring to each of the path in the network.

We simulate a cluster of 768 GPUs. We have 16 spine switches and 24 leaf switches fully connected. Each leaf switch has 4 hosts connect to it. Each host has 8 GPUs and 8 NICs. All the network links and NICs are 200 Gbps. The oversubscription of the network is 2, which is identical to our testbed setting. Our flow-level simulator assumes per-flow fairness. For the workload and job arrival pattern, we adopt a similar setting as the distributed data-parallel deep learning experiment in NetHint [33]. We run 50 jobs of ResNet-50 of model size 100 MB in each experiment. The job sizes are either 16 or 32 GPUs with equal probability. We consider two types of job placement. Random placement means the simulator allocate randomly GPUs to a job. Compact placement means the simulator assigns GPUs that belong to the same rack to a job whenever possible. The jobs arrival follows a Poisson distribution with the lambda set to 200 ms. We run each experiment 5 times and report the average speedup for each job's AllReduce completion time.

Figure 4.10 shows the CDF of performance improvement of using MCCS compared to using NCCL. The numbers in the legend and the vertical dashed lines represent the average speedups across jobs relative to random ring. For random placement, OR and OR+FFA speed up the collective communication by 2.6x and 3.3x compared with using random rings.

With flow assignment, each job can maximize the utilization of the inter-rack network bandwidth. This is because for each network path, we assign a ring to utilize it. Without FFA, the flows within a job can collide on the same physical path. In compact placement setting, OR and OR+FFA still outperform random ring by 3.3x and 3.4x. However, FFA does not add much to OR because the job almost never span more than two racks, and the link capacity of even a single path between two racks would suffice the traffic demand. We observed that the schedule computation takes within 1 ms on average for a job size of 32 GPUs and scales linearly with the job size. The rescheduling occurs only when a job joins or exits.

4.7 Related Work

Integrating collective communication into the network. There are several prior efforts in integrating collective communcation into the network. ATP [133], SwitchML [244], and PANAMA [65] propose offloading AllReduce operations to in-network hardware to enable multi-tenant distributed machine learning. The key difference is that MCCS targets at the public cloud environment, where these works all require tenant applications to be trusted. In these works, a misbehaving or malicious application can circumvent the cloud provider designed collective communication strategy, and this will require well-behaving tenants to adjust their strategies accordingly. In MCCS, all the collective operations are managed through the MCCS daemon. Another difference is that MCCS's performance gain is not from in-network gradient aggregation but from dynamic adjustment to collective communication strategy.

Exposing public cloud network information for tenants to pick collective communication strategies. A separate line of work focuses on letting tenant acquire information about the physical network of the cloud provider in order to pick collective communication strategies. NetHint [33] presents an approach that the cloud provider periodically exposes a hint, containing a subset of the physical network topology and link utilization, to help the tenant pick collective communication strategy. However, a cloud provider may have security and pri-

vacy concerns of exposing their physical network topology and network utilization to cloud tenants to prevent adoption. PLink [161] and Choreo [132] let tenant applications measure their VM-level network bandwidth in order to pick collective communication strategies or decide on job scheduling. These approaches are not guaranteed to be accurate, because reverse engineering the network configurations from a single tenant's observation is generally hard. Further, in both approaches, tenants are making their own decisions on collective communication. In comparison, MCCS controls all tenants' collective communication.

Choosing collective communication strategies based on network topology and bandwidth. Optimizing collective communication strategies for particular network topology and bandwidth configuration is a standard task for developers running large-scale workloads on supercomputers [46, 119]. For machine learning workloads, several prior works have focused on improving collective performance [208, 282, 249]. These works all focus on the single-tenant scenarios. Our work focuses on the multi-tenant public cloud setting. We need to deal with challenges of dynamically changing collective communication strategies, which is not a concern in single-tenant scenarios.

Quality of Service (QoS) in a multi-tenant network. How to let multiple tenants share a cloud network with QoS guarantees is an old topic. A cloud datacenter network often uses a combination of congestion control [229, 4, 315, 79], load balancing [3, 121, 302, 198], and various types of rate limiting techniques [12, 106, 131, 136, 216, 5]. These works focuses on how to share bandwidth given a set of point-to-point network demand. The optimizations MCCS addresses is on having multiple collective communication operations share the bandwidth by selecting collective communication strategies (e.g., the ordering of nodes in an AllReduce ring for each tenant), which is a different and complementary problem.

4.8 Summary

This chapter explores a new service-based approach to collective communication called MCCS. MCCS allows a cloud provider to select collective communication strategies for cloud

tenants and enable the cloud provider to enforce QoS policies on collective communication operations. Collective communication strategies selected by the cloud provider improves tenant performance because the strategies is picked with the knowledge of the underlying cloud network characteristics (i.e., topology, utilization) and can adapt when network characteristics changed. Our testbed and simulation-based evaluations have shown that MCCS improves tenant collective communication performance by up to 2.4x compared to state-of-the-art collective communication libraries, while adding more management features including dynamic adjustment of collective communication algorithm, quality of service, and network-aware traffic engineering.

5. JellyBean: Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures

In Chapter 3 and Chapter 4, we rearchitect lower-level communication abstractions using a service based architecture to improve the performance and manageability of distributed workloads. Next, we step up to application deployment perspective and show how can a system manages and optimizes the deployment of workloads, reducing their deployment costs. In this chapter, we focus on machine learning inference workflows. We present JellyBean, a system that optimizes and serves these inference workflows on heterogeneous infrastructures across edge and cloud. JellyBean aims to optimize the total serving cost of a workflow, given user specified service-level objectives like throughput and accuracy.

5.1 Introduction

Let's start by considering an example ML inference workflow and how to deploy it on heterogeneous infrastructure.

Example. Consider the visual question answering (VQA) workflow in Figure 5.1 for the query "*Who is at the front door?*". The workflow uses multiple ML models for feature extraction and model inference. The infrastructure includes edge devices (e.g., cameras) as well as cloud datacenters. To deploy ML workflows on heterogeneous infrastructures, the following decisions must be made:

- *Model selection.* With advances of AutoML and model compression techniques (e.g., pruning, quantization [100, 243]), each ML operator in the workflow[1] can use various structures or hyperparameters; e.g., the speech recognition operator in Figure 5.1 may use the `base` variant for a faster execution or `large` for a better accuracy. To provide a viable accuracy-efficiency tradeoff, picking individual models in the workflow is non-trivial.
- *Worker assignment.* Each operator must be assigned to a worker for execution. Figure 5.1 demonstrates two execution plans - placing compute near the data source to

[1] Workflows are generated using a standard parser [110] or a natural language interface [124], which are orthogonal to this chapter. See Section 5.3 for more details.

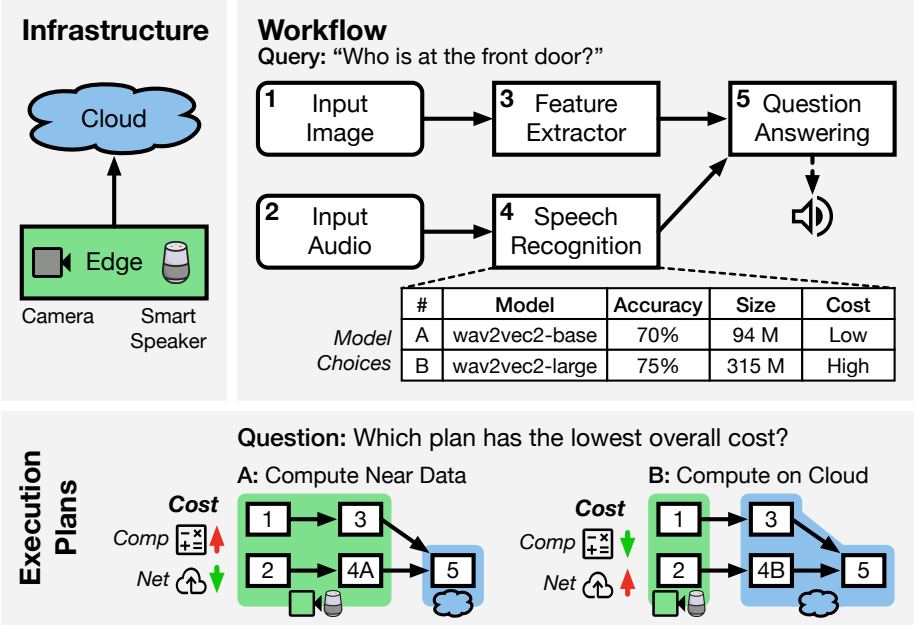


FIGURE 5.1: An example ML workflow of VQA on heterogeneous infrastructures. Different execution plans result in different serving costs, i.e., compute and network.

reduce communication, or moving them to the cloud to take advantage of more powerful (and likely cheaper) compute resources. Choosing an appropriate plan depends on resource availability and costs.

Goals, challenges, and prior solutions. Given the ML workflow, resource availability, input throughput, and target accuracy, we aim to optimize the total serving costs that consist of both compute and networking. It is easy to see that model selection and worker assignment formulate a complex search space.

Current ML serving platforms such as Ray [183], Clipper [41], PyTorch [207], and Spark [299] focused on homogeneous infrastructures (namely cloud datacenter environments). Unfortunately, ignoring resource heterogeneity (e.g., compute, network) often leads to sub-optimal deployments and even feasibility issues given the infrastructure constraints (e.g., on links shared among many high data rate sensors like video cameras). Some prior systems solve this problem in an ad-hoc manner for specific ML workflows, individual models, and fixed infrastructure configurations [41, 108, 146, 301, 250, 237, 1, 207]. Chameleon [108] considers video analytics with one model on a single GPU; Nexus [250]

considers workflows on a homogeneous GPU cluster with no model choices. To our best knowledge, there is currently no off-the-shelf system that optimizes the deployments of ML workflows on heterogeneous infrastructures. As a result, users often manually determine how to best deploy ML workflows.

JellyBean ideas and approaches. We address some initial problems for optimizing ML workflows on heterogeneous infrastructures, and propose a system JellyBean. Given an ML workflow and specifications of the infrastructures, the JellyBean optimizer quickly finds a cost-efficient execution plan with model choices and worker assignments using the following insights:

First, we formulate the problem within a cost-based optimization [32], minimizing the compute and network costs while meeting the input throughput and accuracy constraints. However, optimizing ML workflows poses novel challenges. In the above example, even though we can profile the accuracy and cost for every single model, understanding how different models interact for estimating the overall query accuracy is non-trivial. We leverage a simple but effective model profiling strategy that relies on sampled measurements of interactions between models to estimate query accuracy.

Next, simultaneously solving for optimal model choices and worker assignment is NP-hard and results in an exponentially large search space. We reduce the search space and provide a fast query optimization by (1) making two simplifying assumptions that hold for many real-world scenarios, and (2) identifying key parts that are amenable to greedy approaches. Our evaluations in Section 5.6 show the efficacy in practice.

Lastly, to serve and optimize ML workflows on heterogeneous infrastructures, a flexible runtime is critical such that *the optimizer may explore plans in which models are placed in different workers and locations*. Due to the lack of an existing system to support this, we implemented the JellyBean processor upon Naiad [185] and Timely Dataflow [270], modifying them to enable operator-level parallelism – each worker may handle a subset of the overall workflow. Such a processor and optimizer decide *where to run what*; for *how* to execute each individual operator, we use a containerized runtime with virtualization and ML compiler

techniques [280, 34] such that JellyBean can cope with the infrastructure heterogeneity.

We performed experiments on various real-world use cases, including the Nvidia AI City Challenge [2] and Visual Question Answering (VQA) [6]. Compared with running the ML workflows (1) with all data pushing to the cloud, (2) with all computations staying on the edge, and (3) with optimizations carried out by several worker assignment heuristics, better assigning different parts of the workload to different infrastructure is significantly more effective. We also compared with a few recent ML serving platforms and found that JellyBean is significantly better to achieve the user-specified query-level goal. JellyBean achieves close to equivalent performance compared with an exhaustive brute force search on a small-scale experiment and can still generate efficient physical plans when brute force is infeasible on larger-scale experiments. JellyBean can reduce the total serving cost for VQA by up to 58.1%, and for vehicle tracking in AICity by up to 36.3% compared to the best baselines. JellyBean also outperforms prior ML serving systems (e.g., Spark on the cloud) up to 5x in total serving costs. We have open sourced our prototype: `https://github.com/libertyeagle/JellyBean`.

Contributions of this chapter can be summarized as follow:

- The JellyBean optimizer to derive highly effective execution plans for complex ML workflows on heterogeneous infrastructures given the infrastructure constraints and model choices.
- A flexible JellyBean processor based on a graph dataflow to execute the optimized plans and enable operator-level parallelism on heterogeneous infrastructures.
- Evaluations on real datasets show significant performance improvements over state-of-the-art ML serving platforms as well as running the workflows using heuristics.

5.2 Background

Serving ML on Heterogeneous Infrastructures. These ML workflows show that many application scenarios have input data injected from edge devices. To deploy ML workflows upon these inputs, one way is to put them in cloud datacenters. Clearly, this can often be

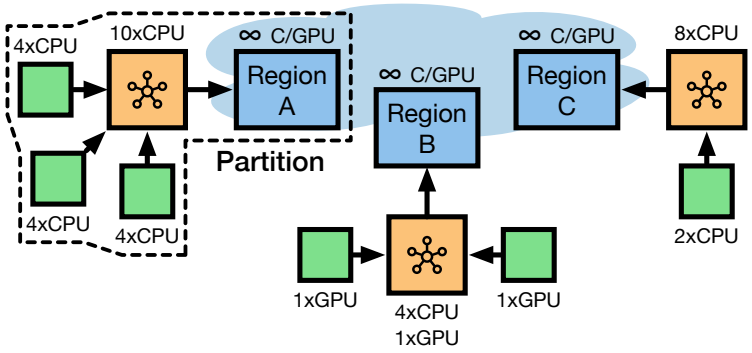


FIGURE 5.2: Deploying ML workflows on heterogeneous infrastructure requires designing physical plans for different partitions.

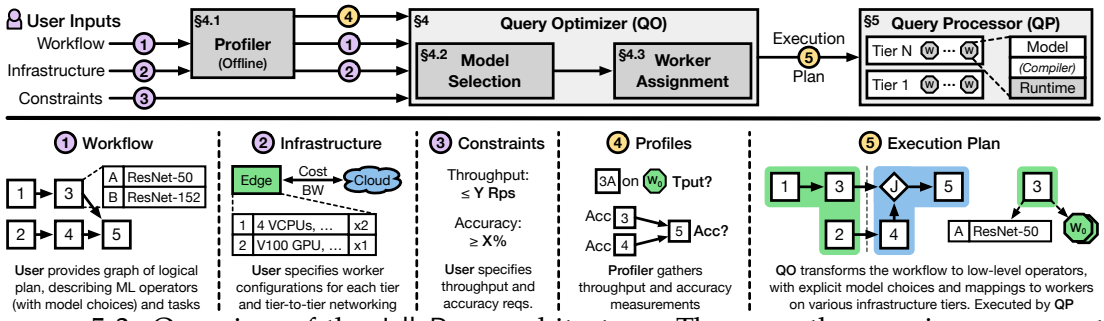


FIGURE 5.3: Overview of the JellyBean architecture. There are three main components: Profiler, Query Optimizer (QO), and Query Processor (QP).

suboptimal since raw inputs (e.g., images and videos) can be large and data movement can be costly.

Moving compute to near the data source is a well-known technique in the big-data systems literature and has been proven to be effective in many use cases [88, 222]. However, today developers still have to hard code or manually tune the physical execution plans for each ML workflow depending on the amount of resources on the edge and costs of various types of resources [108, 170, 117]. We believe this manual approach cannot scale with the rapid development of edge data centers and IoT devices.

Figure 5.2 shows a cloud with three regional datacenters, several local hubs, and edge compute devices. A different execution plan is needed for each partition. For example, different local hubs can have different numbers and types of workers. The cost of running models at different locations can also be different, depending on the cloud region and

Table 5.1: Comparing current ML systems. MS: model selection. WA: worker assignment.

System	Parallelism	QO		Usage	Heterogeneity	
		MS	WA		Worker	Infra.
PyTorch [207]	Data	×	×	Both	×	×
TF [1]	Data	×	×	Both	×	×
Spark [299]	Data	×	×	Infer	×	×
Clipper [41]	Data	×	×	Infer	×	×
Ray [183]	Data, Model	×	×	Both	×	×
Optasia [159]	Data, Op	×	✓	Infer	×	×
Pathways [13]	Data, Model	×	×	Train	✓	×
Llama [238]	Data, Op	×	✓	Infer	✓	×
Scrooge [86]	Data, Op	×	✓	Infer	✓	×
JellyBean (Ours)	Data, Op	✓	✓	Infer	✓	✓

the resource availability at local hubs. We use the term *partition* to denote the tiered infrastructure where different locations within a tier have similar resources. If a partition contains multiple local hubs, they must have similar worker configurations. JellyBean can be used to generate a physical plan per partition.

ML Serving Systems. In order to partially move the ML workflow to the edge devices, besides being able to break it into modules or operators, another necessary condition is a serving system that supports operator-level parallelism on heterogeneous infrastructures. Prior ML systems focused on data, model (i.e., breaking large DNNs into operators) and operator (i.e., breaking workflows into operators) parallelism on homogeneous infrastructures [41, 207, 159, 183], or on heterogeneous workers within a datacenter [238, 86, 13]. We present a qualitative comparison in Table 5.1. Recently, Google's Pathways [13] has started to investigate operator-level parallelism for training large deep neural networks with hybrid cloud infrastructures of CPUs, GPUs, and TPUs. There still lacks an off-the-shelf system for serving and optimizing ML workflows with model choices on heterogeneous and especially IoT infrastructures. We provide a more detailed comparison with related systems in Section 5.8.

5.3 Overview

We discuss our JellyBean design and scope in this section.

System scope. JellyBean aims at serving and optimizing ML inference workloads that can be decomposed into multiple operators deployed on heterogeneous infrastructures. We target infrastructures that exhibit resource heterogeneity across tiers and resource homogeneity within a tier. JellyBean operates over an infrastructure configuration that describes a single partition of a potentially larger infrastructure. The optimization takes into account input throughput, resource cost, availability and efficiency, and targets scenarios in which compute and communication are important factors in the total serving cost. The JellyBean processor provides a flexible runtime and decouples resource heterogeneity using a containerized runtime with virtualization and ML compilers, hence targeting a wide spectrum of edge and cloud devices.

System overview. In Figure 5.3, we present an overview of our JellyBean system architecture and the workflow for processing an ML workflow. There are two main components: the query optimizer (QO) and the query processor (QP). The query optimizer generates an execution plan for the ML workflow, while the query processor runs the execution plan across heterogeneous infrastructure.

JellyBean takes the following inputs:

- *Workflow.* Each input workflow is a directed acyclic graph (DAG) with compute operators on the nodes and input-output relationships between operators on the edges. The operators can be ML models or relational operations. Declarative queries can be parsed into workflows [110, 124] as is done in [160, 116].
- *Model choices for each ML operator.* Each ML operator may use different models with the same semantics but different structures or hyperparameters. These models have different accuracy and cost profiles. JellyBean may profile these models offline if necessary.
- *Infrastructure specifications.* We consider infrastructures that consist of heterogeneous

resources (i.e., compute, storage and networking) in multiple tiers - each tier is a group of efficiently interconnected resources that share common specifications.

- *Input throughput and target accuracy.* Users provide a target accuracy on the query output; meanwhile, JellyBean must keep up with the input throughput. The target accuracy restricts the model selection to generate a low-cost physical plan.

Our query optimizer generates the physical plan in two steps. First, it selects models that satisfy the target accuracy with the least costs (Section 5.4.2). Here we do not have worker assignments yet, so the exact costs of deploying the selected models are unknown. We approximate the costs based on the characteristics of the models (e.g., model sizes, the latency of inference on a standard CPU/GPU) and use beam search to select the best K configurations. Each configuration includes the model selection for all models in the workflow.

The second step is to determine the worker assignment (Section 5.4.3). We again use a beam search method. We progressively determine the worker assignment by choosing a set of workers for each operator to achieve the lowest compute and networking costs. More than one worker may be assigned to an operator to consolidate the costs. The best worker assignment is derived then for each of the K configurations and choose the best physical execution plan for both model selections and worker assignment.

The JellyBean processor is a distributed query processing engine upon Naiad [185] and Timely dataflow [270] to provide a low-overhead dataflow abstraction. However, Naiad and Timely Dataflow use a homogeneous datacenter setup with data parallelism only JellyBean augmented their codebase to incorporate operator-level parallelism, allowing different workers to run different portions of the workflow. Each worker leverages a containerized runtime with virtualization or ML compilers [34, 194] to offset heterogeneity (Section 5.5).

Table 5.2: Set of common notations used in our description.

	Notation	Definition
User Input	G	Graph of logical plan ($G = \langle V, E, M, m \rangle$) Vertices V, Edges E, Models M Model Choices $m : V \rightarrow \mathcal{P}(M)$
	I	Set of infrastructure tiers
	W, W_i	Set of workers overall [or for tier $i \in I$]
	C_B	Worker-to-worker communication cost ($C_B : WxW \rightarrow \frac{\$}{\text{byte}}$)
	T, T_v	Input throughput overall [or for node $v \in V$]
	A	Target overall accuracy
Profiler	C_C	Unit compute cost for model on worker ($C_C : MxW \rightarrow \$$)
	t_u^w	Throughput for model u on worker w
	r	Unit input size at v from u ($r : VxV \rightarrow$ byte)
QO	s	Model selection ($s : V \rightarrow M$)
	a	Worker assignment ($a : V \rightarrow \mathcal{P}(W)$)

5.4 Query optimizer

5.4.1 Problem formulation

We consider our infrastructure to be composed of a number of workers with diverse computing capability distributed across multiple tiers (e.g., edge, hub, and cloud). Data sources are located on the lowest tier (i.e., W_1), often with some limited compute resources. Workers on higher tiers tend to have more computing capability but are far away from the data sources. We assume a set of workers W, which are partitioned into $|I|$ tiers.

Let the input of our optimizer be a logical plan graph G in which each node $v \in V$ corresponds to an ML or regular relational operator. For each ML operator, the user specifies a list of candidate models $m(v)$, each having a different accuracy and runtime performance. These models can be developed independently or can be variants of other well-known models through quantization [51, 100], distillation [69], and pruning [155, 243]. Section 5.7 discusses techniques to generate a diverse set of model choices. A model's accuracy and performance can be either provided by the user or profiled by JellyBean. We use $s(v)$ to denote the model choice for v. Meanwhile, we assign for each logical operator v a list of workers $a(v)$ in the heterogeneous infrastructure. The infrastructure specification

contains sets of each type of worker a tier has, the cost of each type of worker, and the communication costs between different tiers. Note here our formulation only considers a single partition. This is because each partition (shown in Figure 5.2) requires a different physical execution plan. Table 5.2 illustrates the notations used in this chapter as well as inputs to our query optimizer. Note that the compute and communication costs here as unit monetary costs; the former is the hourly price per worker, and the latter is based on network traffic (i.e., data movement on the DAG edges).

We aim to solve worker assignment[2] $a : V \rightarrow \mathcal{P}(W)$ and model selection $s : V \rightarrow M$ simultaneously, such that the overall query accuracy (acc) is beyond a user-specified target A, and that the system's throughput ($t_{v_{out}}^{a(v_{out})}$ at the output node v_{out}) is no less than a target T. We describe our target cost function and our query optimization as:

$$\arg\min_{a,s} \sum_{v \in V} \sum_{w \in a(v)} C_c(s(v), w) \; + \tag{5.1}$$

$$\sum_{(u,v) \in E} \sum_{\substack{(w_u, w_v) \in \\ a(u) \times a(v)}} C_B(w_u, w_v) R(u, v)$$

$$s.t. \;\; acc \geqslant A, \;\; t_{v_{out}}^{a(v_{out})} \geqslant T,$$

where $R(u, v)$ denotes the consumed network bandwidth from u to v. The formulation above minimizes the ML workflow's combined compute (first term) and networking (second term) costs and is NP-hard, because the sub-problem of solving only the worker assignment is already a combinatorial optimization that can be reduced to a binary knapsack problem (which is NP-complete [64]).

Assumptions. We make two assumptions in our optimization to reduce the problem complexity without losing generality, as these assumptions hold for many realistic use cases:

- `A1`: We assume that communication costs $C_B(w_1, w_2)$ to have the following properties: 1) set to 0 if w_1 and w_2 are on the same infrastructure tier and are in the same location,

[2] We note that assigning for each operator a list of workers is equivalent to picking the model to execute for each worker.

2) otherwise set to a positive value. This is common in many use cases, as workers in the same tier either do not inter-communicate (e.g., among edge devices at different locations) or use high-speed networking (e.g., among datacenter nodes) with negligible costs.

- A2: We assume that all workers only communicate with peers in the same infrastructure tier or any higher tier, thus making information flow in one direction[3]. This assumption implies that for all edges $(u, v) \in E$, the set of workers $a(v)$ are all on tiers greater than or equal to the highest tier of any worker in $a(u)$. This is reflected in C_B by values of $+\infty$ for pairs of workers that violate this one-way flow assumption.

Model Profiling. JellyBean needs to understand the impact of selecting different models on accuracy and throughput in order to meet the constraints specified by the user for the overall workflow. While users can optionally specify the accuracy and performance of models for different infrastructure workers, JellyBean supports automatic profiling using validation datasets provided by the user. If a worker cannot run a particular model (e.g., model requires a GPU but the worker is CPU-only), we set both the accuracy and the throughput to be zero. Otherwise, JellyBean measures the runtime performance in terms of the throughput for the model on every worker type in the infrastructure. Note that we use the mean throughput of each model (and thus compute cost) relative to the input throughput during cost calculation, since operators in ML workflows may have different output-to-input ratios. For model accuracy, we need to understand the accuracy response of a model with respect to the accuracy of upstream models whose outputs are fed into it. JellyBean varies the input accuracy by selecting different upstream models (with different accuracy profiles) and measures the output accuracy response of the model under test. For example, consider a model with two inputs and exhibits the following accuracy profile: $(60\%, 50\%) \rightarrow 55\%$, $(50\%, 60\%) \rightarrow 60\%$, $(70\%, 90\%) \rightarrow 65\%$. This profile enables us to conservatively estimate the output accuracy by identifying the row that is closest to (but

[3] We note that the final result from the workflow may be transferred back to the lowest tier (e.g., user's device), but we do not model this.

not higher than) the accuracy of all inputs; for example, if the input accuracy is (55%, 83%), then we can conclude the output accuracy is at least 60%. One assumption we make here is that the output accuracy is monotonically increasing with respect to each input accuracy (with the others fixed). In Section 5.4.2, we demonstrate how we use the accuracy profile to select models that satisfy the user's target end accuracy.

Next, we describe our solution that finds highly effective execution plans as well as components to derive query-level accuracy and, assign workers across the heterogeneous infrastructure.

5.4.2 Model Selection

Model selection balances the inference cost and model accuracy:

Satisfying Accuracy Constraints. One challenge in our model selection is to estimate the query-level accuracy given profiles of individual ML models, which can be non-trivial due to the dependencies among them. So far, this has not been discussed in any prior work, and we propose a solution here as follows.

We consider the dependency graph of the ML operators in the logical plan G. For each operator, we can assign (choose) a model variant; the final accuracy for the model selections s should satisfy a user specified accuracy threshold A. We use the model profiles to determine whether a model configuration satisfy the accuracy constraint. In each model's accuracy profile, we need to choose a row such that the output accuracy of a model is larger than a downstream node's required input accuracy. Also, the final output model's accuracy has to be above the target end-to-end accuracy.

Reducing the Total Cost. Another problem during model selection is that we do not know worker assignments yet and thus we cannot use a concrete cost. Thus, we need to choose models based on a different cost definition. We can use the execution latency on a single GPU or the number of parameters in the model. In our current prototype, we use a simple notion of cost: the latency for model inference on the most powerful infrastructure worker (e.g., NVIDIA V100 GPU in our evaluation).

We use the accuracy profiles and perform a *beam-search* to find the model assignments that can attain user's specified end-to-end accuracy threshold. We traverse the graph in reverse topological order, and assign the model for each node. Each candidate is a combination of partial model assignment and the accuracy requirements for upstream nodes. Specifically, we first extract the accuracy requirement for a node that we are currently assigning, and then iterate through all the candidate models for the node and find models whose output accuracy is greater than the threshold from the downstream models. When there are multiple models satisfying the output accuracy, we pick the ones that have the lowest cost. There can be many model configurations that satisfy the accuracy constraint, and we maintain the best B_{MS} number of model configurations based on their costs. After one model selection is found for this node, we then update the model assignment to propagate the accuracy constraints to upstream nodes until all nodes have a model assignment. We have to maintain more than a single candidate model configuration because our cost estimation can be not accurate. The real cost should be the actual cost of deploying this model on a particular worker type in the edge or cloud; here we simply use the latency or model size as the cost.

5.4.3 Worker Assignment

The goal here is to take the set of candidate model selections from the previous step and determine the best mapping from models to available infrastructure workers that minimizes the overall cost while meeting the input throughput to our system. We will first present an overview of our worker assignment algorithm, which makes greedy choices along two dimensions to reduce the large search space for worker assignment: 1) the order of assigning nodes $v \in V$ to workers, and 2) the workers $w \in W$ to be assigned. Next, we will describe our approach for determining the per-input cost of assigning the execution of a model to a given worker, which enables our greedy selection of workers. We also discuss key refinements that improve optimality in practice.

Computing Assignments. We present our solution in Algorithm 1. We consider as input a

specific candidate model selection (out of the top-K candidates produced by the previous phase). The output consists of a mapping between nodes in the logical graph and sets of available workers.

Algorithm 1: Worker assignment.

Input : Model selection $s : V \rightarrow M$
Output: Worker assignment $a : V \rightarrow \mathcal{P}(W)$
Function *Avail(W, a, i)* → *Returns unassigned workers in tier* $\geqslant i$
Function *MinCost(W, s, v)* → *Returns worker with min cost (Eq 5.2)*
Function *TCoeff(w)* → *Returns throughput coefficient based on tier*
Function *Top(a, k)* → *Returns top-k best assignments in set*

1 $a_B = \{\varnothing\}$ // Current set of assignments in beam
2 **for** $v \in Topo(V)$ **do**
3 $a'_B = \{\}$ // Next set of assignments in beam
4 **for** $a_b \in a_B$ // Iterates over current set of assignments in beam **do**
5 **for** $i \in I$ **do**
6 $T_{\text{rem}} = T_v, a_{cur} = a_b$
   // Greedily assign workers up to throughput req.
7 **while** $T_{rem} > 0$ **and** $|Avail(W, a_{cur}, i)| > 0$ **do**
8 $a_{cur}[v] \cup= \text{MinCost}(\text{Avail}(W, a_{cur}, i), s, v)$
9 $T_{\text{rem}} -= (t_v^w \times \text{TCoeff}(w))$
10 **if** $T_{rem} \leqslant 0$ **then** $a'_B = a'_B \cup \{a_{cur}\}$
11 $a_B = \text{Top}(a'_B, B_{WA})$ // Keep only top assignments in beam
12 $a = \text{Top}(a_B, 1)$

In Line 2, we start by iterating over each node $v \in V$, using a topological ordering such that parent nodes are assigned before their downstream child nodes. While an optimal solution would need to consider the assignment of all nodes jointly, this is computationally intractable. However, due to the nature of realistic workflows and our assumption `A2` that limits communication in one direction between tiers (i.e., from lower to higher), greedily computing worker assignments based on the topological ordering is a reasonable approximation. For any particular V and E, there may be many valid topological orderings; therefore, we extend our approach to also iterate over a constant number of different, randomly-selected topological orderings to improve the optimality.

For a given node v, we need to assign a set of workers to execute ML operator (or

task), such that we limit the cost while meeting the input throughput. Each worker can be assigned to a node v[4], and such assignments formulate a combinatorial optimization which is NP-hard [64].

We use a greedy approximation for worker assignment by considering the cost of assigning a worker w to handle the execution of node v (with the assignment cost defined at the end of this section). We assign workers based on availability (i.e., not already assigned) and ordering from lowest to highest cost until the input throughput is met, or until we run out of workers to assign (Lines 7-9). Given our assumption of one-way communication between infrastructure tiers (`A2`), if a node u is greedily assigned to a worker on a higher-tier, then all nodes $v \in V$, where there exists an edge from u to v, are unable to be placed on lower tiers. We modify this by computing the greedy assignment over expanding pools of available workers, where the number of pools is equal to the number of tiers $|I|$ and the i$^{\text{th}}$ pool contains all workers in the i^{th} tier or lower. We use a beam search to reduce the search space by keeping the best B_{WA} candidate assignments (i.e., those with the lowest cost) out of the $B_{WA}|I|$ considered at each step (Line 11).

Since each tier may be distributed among one or more locations, we cannot simply consider the remaining throughput based on that achieved by a candidate worker w for node u (i.e., t_u^w). Instead, we need to multiply this by the TCoeff(w), which computes the factor based on the number of locations from the tier of w up to the root of the partition (e.g., cloud tier). Consider an example infrastructure that consists of the cloud, hub (2 locations), and edge (5 locations); TCoeff(.) is 1, 2, and 10 for workers on the cloud, hub, and edge (respectively).

Assignment Cost. To greedily pick workers with minimal unit (or per-input) cost, we need to take both computation and communication costs into account. Considering the cost for a node $v \in V$, with model selection s, running on a worker w, our overall cost equation is:

$$C_C(s(v), w) + \sum_{(u,v) \in E} \sum_{x \in a(u)} C_B(x, w) \left(\frac{t_u^x}{T_u} \right) r(u, v), \tag{5.2}$$

[4] We use one-to-one mapping due to the low overhead of our processor. See Section 5.5 and Section 5.7.

containing the unit cost for computation (first term) and communication (second term). $s(v)$ is the selected model out of all choices for node v, and the unit computation cost is derived from the profiler using the cost of each worker and the throughput of the worker while executing the selected model.

For the unit communication cost, we leverage all previous assigned nodes $u \in V$ that have edges to the current node v. Hence, the second term involves summing the costs across all workers assigned to u (i.e., $x \in a(u)$) and the worker w that is being considered. Note that we only consider the parents of v and not it's children, since our greedy algorithm operates in the topological ordering of the nodes, such that the assignments $a(u)$ for all child nodes u are already known. If w_u and w are on the same tier, the communication cost between the workers will be zero (`A1`); otherwise, there is some bandwidth-based cost for the traffic between the infrastructure tiers for x and w. This bandwidth cost is multiplied by the amount of communication for w_u, which is based on the unit input size $r(u,v)$ and the fraction of that input which is handled by x. The fraction of input is equivalent to the ratio of the throughput for u on x compared to the input throughput T_u. For instance, a node v takes inputs from u that is assigned to an edge worker x_1 (40% inputs) and a cloud worker x_2 (60% inputs). If we assign a worker at the cloud, the communication cost has to include the split linkages. The term t_u^x / T_u is the fraction of the $u \rightarrow v$ traffic contributed from x.

5.5 Query processor

We prototype JellyBean upon Naiad [185] and Timely Dataflow [270] code base, which offered a low-overhead dataflow abstraction. However, there are additional features that JellyBean requires. We outline the challenges and our implementations in the following.

Operator-level parallelism. Timely Dataflow is designed for data parallelism. Instead, JellyBean aims for operator-level parallelism, spanning the workflow and compute nodes across different workers; hence they can execute different portions of the plan. The challenges here are two-fold: (1) all workers in Timely Dataflow must execute the same set of operators

with different data inputs; (2) Timely Dataflow uses all-to-all communications for progress tracking, causing unnecessary overhead.

In the prior sections, we described our optimizer to assign workers to operators, where each worker is responsible for one operator in the graph. Indeed, executions of pipelines or workers that are assigned with multiple nodes are used in production database systems [202]. Our solution is simple but effective; as our experiments will show, we may put multiple workers on a single device, since the compute and network overhead of our processor is low.

Therefore, each worker only acquires its input data from upstream workers and sends its outputs to the downstream workers. We build a relay mechanism to serve as a "broker" between adjacent workers. There can be one or more relays in each worker; each receives input data from the relay nodes in the upstream workers. It also collects the outputs and sends them to the relay nodes in the downstream worker. To implement this, we use a thread for each upstream worker that keeps pulling data from the upstream worker's relays through TCP streams and maintaining proper buffers. There is also a thread for each downstream worker that pulls output data and sends it to the relays of the downstream workers. In such a manner, operator-level parallelism is achieved by properly parallelizing independent workers (which can be on the same device), tracking their progress, and syncing by treating each worker in our compute graph as a Naiad node. Lastly, we modified the progress tracking algorithm to support node-to-node progress updates.

Networking protocols. Timely Dataflow supports communication among the worker nodes only by relaying on the master node; this results in unnecessary data movements. We augment the networking protocols to enable peer-to-peer communications among the workers; a low networking overhead is essential in a dataflow engine that supports operator-level parallelism.

Containerized worker runtime. Timely Dataflow supports homogeneous runtimes only. To offset runtime and hardware heterogeneity in JellyBean, each compute node deploys a containerized runtime with a Linux virtual machine to hold one or more Naiad workers. Table 5.3 illustrates part of the operators and models used in our experiments; each may

Table 5.3: Some AICity models/operators used in our experiments.

Model	#Parameters (Millions)				
resnet	**18**	**34**	**50**	**101**	**152**
Object Re-identification	11.7	21.8	25.6	44.5	60.2
YOLO	**v5n**	**v5s**	**v5m**	**v5l**	**v5x**
Object Detection	1.9	7.2	21.2	46.5	86.7
wav2vec2	**base**	**large**			
Speech Recognition	94.4	315.5			

contain a feature extraction or classification model. Within each container, JellyBean optionally applies ML compilers [194, 34] to adapt the model assigned by the QO to the worker hardware. By default, the ML models are implemented in PyTorch within the Naiad map functions.

Relational operators support. Timely Dataflow did not support relational operators including filters, join and group-by-aggregation upon columnar inputs. We hence implement these operators in JellyBean. The metadata is packaged with the data being transmitted in-between the workers to facilitate relational operations.

Remark. The runtime backend of our prototype consists of 12K lines of new Rust code beyond the Timely Dataflow v0.12. While our query optimizer is independent to the runtime engine, supporting broader runtime backends can be interesting future work.

Table 5.4: Four workload and infrastructure setups. We use $m \times n$ to denote that there exists m servers, each has n vCPUs.

Dataset	VQA		AICity	
Setups	Objectives	Infras	Objectives	Infras
`small`: (5 nodes)	Accuracy: 0.55, Throughput: 9 rps.	Edge: 1x4, 1x8, 1x16, Cloud: 2xV100.	Accuracy: 0.65, Throughput: 3.5 fps.	Edge: 1x4, 1x8, Hub: 1x16, 1xV100, Cloud: 1xV100.
`medium`: (9 nodes)	Accuracy: 0.56, Throughput: 40 rps.	Edge: 1x2, 1x4, 2x8, 1x16, Cloud: 1x48, 3xV100.	Accuracy: 0.70, Throughput: 8 fps.	Edge: 1x2, 1x4, 1x8, Hub: 1x8, 1x16, 1xV100, Cloud: 1x48, 2xV100.
`large`: (15 nodes)	Accuracy: 0.56 Throughput: 60 rps	Edge: 2x2, 6x4, 1xV100, Cloud: 3x8, 3xV100.	Accuracy: 0.70 Throughput: 11 fps	Edge: 2x2, 2x4, Hub: 4x4, 2xV100, Cloud: 3x8, 2xV100.
`xlarge`: (30 nodes)	Accuracy: 0.57 Throughput: 100 rps	Edge: 6x2, 10x4, 2xV100 Cloud: 2x4, 6x8, 4xV100	Accuracy: 0.75 Throughput: 20 fps	Edge: 6x2, 3x4, 1xV100, Hub: 8x4, 2x8, 2xV100, Cloud: 1x4, 4x8, 3xV100.

5.6 Evaluation

We evaluate JellyBean against state-of-the-art techniques for machine learning model serving with the following goals.

- G1 Is it beneficial to use JellyBean for serving ML inference workloads on heterogeneous infrastructures? We showcase end-to-end accuracy and cost measurements comparing with relative systems on two real-world use cases.
- G2 We measure the effectiveness and cost overhead of the JellyBean processor on various cloud and physical runtime.
- G3 To show that our optimizer is near optimal, we tease apart the usefulness of various aspects of the JellyBean optimizer in an ablation study and compare with alternative ML model selection and placement strategies as well as lower bounds.
- G4 We study the robustness and flexibility of JellyBean in a sensitivity analysis by varying the systems and workload settings.

5.6.1 Experiment Setup

Datasets. We consider two realistic machine learning workflows (and associated datasets) for model inference:

NVIDIA AI City Challenge (AICity) [2] is a public dataset and benchmark to evaluate tracking of vehicles across multiple cameras. The dataset is divided into 6 traffic intersection scenarios in a mid-sized US city, which in total contains 3.58 hours of videos collected from 46 cameras. A frame has 1.1MP (megapixels) and 22 objects (cars) on average. The ReID models are trained on the CityFlowV2-ReID dataset [267], while the object detection models are pre-trained on the COCO image dataset [151]. We leverage their testing scenario in our system evaluations. Figure 2.2 demonstrates a typical workflow upon this dataset with an object detection model, an object Re-identification (ReID) model and the subsequent tracking modules to derive cross-camera vehicle trajectories.

Visual Question Answering (VQA) [6] is another public dataset containing open-ended questions about images from the COCO image dataset [151]. The task is to generate an

answer (from a large set of candidate responses) for an image-question pair. This dataset has 614,163 questions on 204,721 images. The mean input image resolution is 0.3MP and the mean input speech length is 1.5sec. The validation set from the original dataset split is used in our evaluation. Figure 5.1 demonstrates a typical workflow for VQA.

In our offline profiling, we measure the accuracy of 10 model combinations on the VQA validation set with 121,512 samples, taking 10-20 minutes depending on the model combinations. As for AI City, where test labels are not available, we use the official benchmarking API [2] to get the IDF1 scores. We profile 20 model combinations, and the profiling takes 1-2 hours depending on the model combinations. We also use reported accuracy on standard benchmarks whenever available [297, 223, 89]. We note that these are *one-time, per-database* costs and can be amortized among different ML workflows later on. We use P75 efficiency numbers as input to our optimizer to offset runtime variance; our sensitivity analysis in Section 5.6.4 discusses using other percentiles.

Workload and infrastructure settings. We conduct our experiments on the IBM cloud where the workload and infrastructure setups are detailed in Table 5.4, where we show the input throughput in frame/request per second (FPS/RPS). We use the mean per-frame/audio size from the input dataset in our cost model. We evaluate four setups ranging from `small` to `xlarge` by varying the number and type of available workers for each infrastructure tier as well as the throughput and accuracy targets. Each compute node represents a virtual machine as described in Section 5.5 with the number of vCPUs specified (2-48), while each GPU compute node represents a VM with a 16GB NVIDIA V100 GPU. The memory of each node ranges from 4GB to 192GB and the bandwidth ranges from 3Gbps to 25Gbps. In Section 5.6.4, we show experiments when the bandwidth is limited. While the absolute infrastructure tier configurations may not capture all real-world infrastructure setups (e.g., IoT devices with compute <2 vCPUs), we note that the *relative* compute power difference between tiers does capture this. Using these settings strengths our evaluations as our processor offsets hardware heterogeneity by virtualization and ML compilers (Section 5.5).

We strive to echo real-world scenarios when setting up the base resource costs in our

experiments; nevertheless, there can be orthogonal factors such as dynamic pricing models [77]. Hence, we use the unit compute and networking costs based on the pricing catalog of the IBM Cloud as of April 2022 [94]. The unit costs increases sub-linearly along with the resources used (e.g., 1 and 1.5 unit costs for 2 and 8 vCPUs respectively, and 3 for V100). The communication costs among different tiers (e.g., from edge to cloud) range from 0.1 to 0.3 unit cost per GB; for example, direct communication from edge to cloud bypassing local hubs is more expensive.

We also leverage prior VQA and AICity solutions on top of the benchmarks from [16, 153] and set up the accuracy and throughput targets used in our experiments based on the profiles of these state-of-the-art solutions. The virtual machines are chosen such that `small` and `medium` aim for low serving costs without edge GPUs, while the larger setups aim for low latency with edge GPUs available. The later cases also demonstrate how compute can be moved to the cloud when the edge has not enough compute power.

Evaluation metrics used in our experiment include:

Performance. We report both estimated and actually achieved throughput in one hour, as well as various overheads incurred by our query optimizer and processor. We also aim for a system that provides viable trade-offs between accuracy and throughput; we report the actual accuracy scores on the validation sets described earlier.

Serving costs. We report the compute and networking costs of executing the ML workload on the infrastructures specified in Table 5.4. We evaluate the costs while varying the target accuracy and input throughput. For JellyBean and all baselines (described next), we report the serving costs and other metrics when the system saturates, excluding model loading, system startup and shutdown time.

Baselines and comparisons. To compare JellyBean (`JB`) over state-of-the-art ML serving solutions on heterogeneous infrastructures, we consider the following baselines in our experiments:

Worker assignment strategies. Inspired by geo-distributed database placement [88, 222, 206] and VM placement strategies [71], we compare with using the following model selection

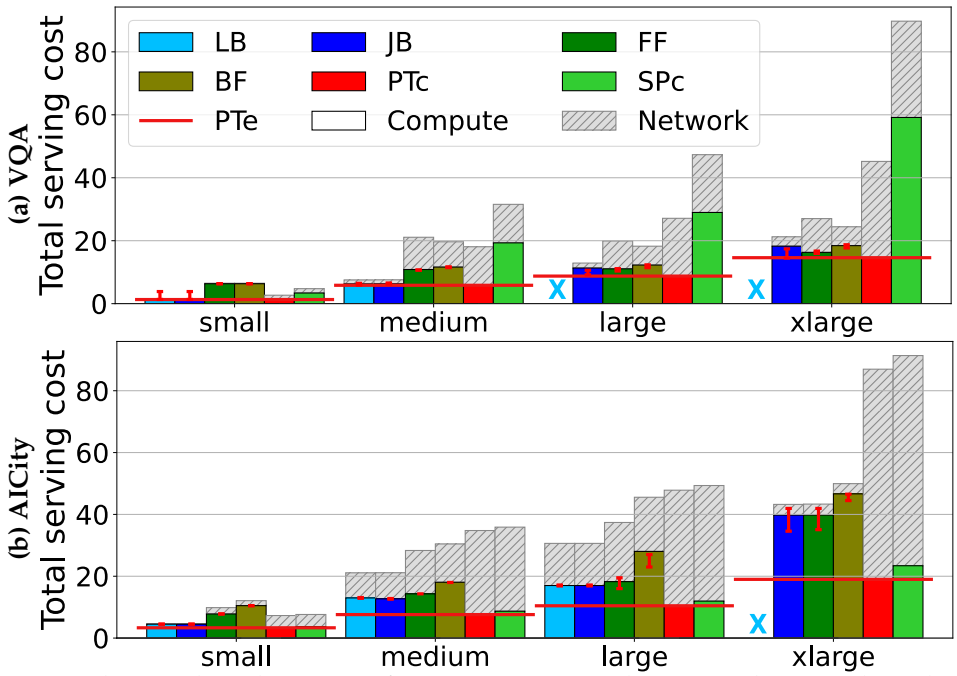
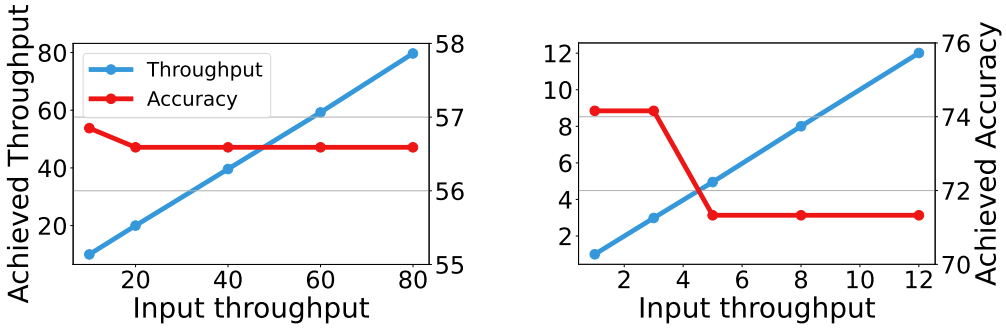


FIGURE 5.4: End-to-end evaluations of ML serving. We showcase the actual total serving costs using the P75 profiles.



(a) VQA (b) AICity

FIGURE 5.5: Achieved throughput and accuracy given different input throughput on the `medium` setup.

Table 5.5: Cost analysis on the AICity dataset.

Model		medium			large		
Select.	Assign.	Comp.	Net	QO	Comp.	Net	QO
JB	JB	12.7	8.4	6.5ms	17.0	13.6	7.8ms
LB	LB	13.0	8.1	2.1**s**	17.0	13.6	27**min**
JB	FF	9.0	14.1	3.9ms	12.0	19.1	5.7ms
JB	BF	16.0	8.3	3.4ms	20.0	13.5	3.4ms

and worker assignment strategies while using the JellyBean processor[5]: (1) Best Fit (BF) is inspired by geo-distributed database optimizers [88, 222] to reduce the networking costs; it uses the most accurate model and greedily assigns jobs to the cheapest worker on the same infrastructure tier. (2) First Fit (FF) follows a classic VM placement strategy [206] in which each operator uses the most accurate model and assign jobs to the cheapest worker regardless of their location. (3) Lower bound (LB): we compute a lower bound of the serving cost by enumerating over all possible model choices and worker assignments when keeping the placement constraints (A2). This baseline showcases the optimality of our solution and it is worth noting that BF and FF may not follow the networking constraints used in JB and LB.

End-to-end ML serving. To our best knowledge, there lacks an off-the-shelf solution for serving ML on heterogeneous infrastructures while supporting the functionalities that JellyBean can provide. We use the following variants of existing systems to echo the real-world ML deployments. (1) We perform all computation on a single GPU worker using native PyTorch to handle the entire workflow. Doing so has the minimum compute overhead from the software stack beyond PyTorch but has to pay potentially large networking costs if the workers are on the cloud. By default, we use the most accurate models that are available and denote PTe as running PyTorch on the edge, *pretending* that there is a V100 GPU and counting the GPU costs; PTc runs PyTorch on a cloud V100 GPU, which is equivalent to PTe plus networking costs. (2) We assume the data is transferred to the cloud and use the

[5] We note that Worst Fit placement [206] that greedily puts models on the most expensive location does not fit in our context.

most accurate models in a Spark. This baseline leverages all the cloud GPU workers in each infrastructure setup (Table 5.4) and performs data parallelism upon native PyTorch wrapped in a map function (`SPc`).

Model selection. The baselines above use the most accurate models available, since none of them solves the model selection problem. We will perform in Section 5.6.3 an ablation study to examine the effectiveness of our proposed model selection strategy, showing the optimality gap from using brute force.

5.6.2 System Evaluations

System efficiency. We showcase `G1` by the the end-to-end evaluations in Figure 5.4 and Table 5.5 using various workload and infrastructure settings in Table 5.4. Note that in Figure 5.4, the error bars illustrate the estimated costs using the P50 and P90 profiles (see Section 5.6.4). 'X' indicates unsolvable given 1h of QO time. For Table 5.5, we show the costs for one hour of input data with input throughput specified in Table 5.4 and the corresponding query optimizing time (QO). We note a few observations here:

`JB` demonstrates the best performance with different datasets and setups compared to the baselines. On VQA, `JB` saves the total serving cost up to 58.1% compared to the best-performing baseline (`PTc`) and up to 5x compared to end-to-end ML systems `SPc`. On AICity, `JB` saves the total cost for up to 36.3% compared to the best-performing baseline (`PTc`) and up to 2.1x comparing to `SPc`.

We showcase the *actual* throughput and accuracy in Figure 5.5. `JB` achieved near 1:1 for actual:expected throughput (diagonal line). The results for the `large` setting is shown in Appendix C. With increasing input throughput but fixed available infrastructures, JellyBean successfully trades off throughput with accuracy by picking suitable models.

Comparing `JB` to `LB`, we observe a subtle difference in the overall serving costs – with different input throughput, 94.2% of the chances `JB` provides a total cost that has less than 1% difference to that provided by `LB` on AICity. `LB` requires a large QO time as will be shown next and becomes unusable – in AICity, `medium` has 8K choices while `large` has 7M

choices.

Figure 5.6 illustrates a qualitative example of the execution plans of `JB` and `LB` when they do not match. JB uses 1x16 worker and a larger ResNet model for feature extraction, while LB uses 1x2 and 1x8 which leads to a lower cost. `BF` and `FF` failed to find overall optimal execution plans in our experiments; though in some cases, they find plans with low compute or low network costs solely (e.g., BF with low network cost while FF with low compute cost in Table 5.5). This can be as expected since their heuristics ignore model accuracy-efficiency trade-offs and the resource availability on heterogeneous infrastructures. In most cases, `BF` and `FF` have much higher costs than `JB`; using heuristics that consider network or compute cost solely is suboptimal. On the other hand, model selection greatly helps to reduce the overall costs, especially when the accuracy target is lower. `PTc` and `SPc` use homogeneous GPU computing, which results in lower compute costs than `JB` yet larger networking costs since the raw data must be transferred from the edge. `SPc` exhibits more overhead as compared with `PTc` [169]. `PTe` is a hypothetical baseline that assumes strong GPUs on the edge, and thus leads to minimum compute costs at zero networking cost. In real-world applications, with no resource constraint, users should adopt this solution; however, this is often not true in practice.

Further evaluations in Appendix E show that JellyBean often yields serving costs equal to or close to the lower bound. We also discuss some failure cases in Section E.2. For instance, in the case that Assumption A2 is removed.

We observe that the runtime variance is low across all setups; for example, the standard deviation from five runs on the `large` setup is 0.003% for AICity and 0.020% for VQA. The runtime variance on the `xlarge` setup is reported .

System overhead. Table 5.5 also illustrates `G3` – the `JB` optimizer has a small overhead with the QO time of `JB` in a few milliseconds. In comparison, LB uses brute force, which incurs adverse QO time in larger infrastructure settings (e.g., 27 minutes for `large`). Other placement strategies have smaller QO time due to a smaller search space, but the total serving costs are larger.

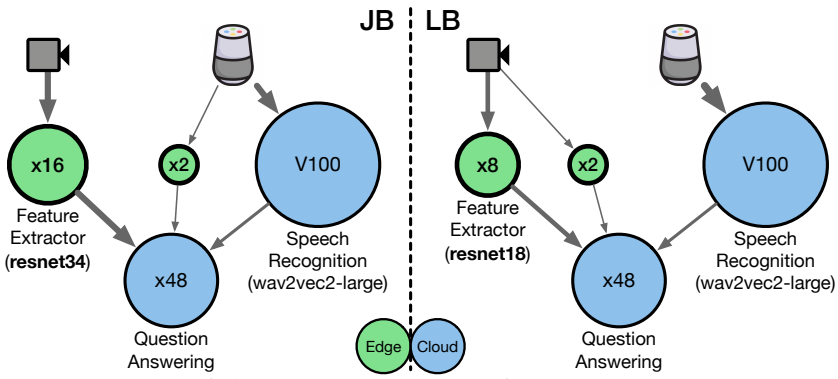


FIGURE 5.6: Comparison of the execution plans of JB and LB on VQA using the `medium` setup modified to 20 rps.

We further demonstrate in Table 5.6 the compute overhead of the `JB` processor, where E denotes Edge, H denotes Hub, C denotes Cloud, Original denotes the latency with native PyTorch, QP exec denotes the overhead of executing the operator in the JellyBean query processor, QP network denotes the overhead of communication. We show the 50th and 90th percentile of various ML operators in native PyTorch and by the `JB` processor. The overhead caused by `JB` processor, as partially been discussed in [185], contains that for metadata parsing, data (un)packing, network I/O, and task scheduling. The QO latency is reported on 1x8 virtual CPU node with a Python implementation. Results indicate a small overhead ranging from a few to 19% upon the native PyTorch executions. This is significantly smaller that that of Spark which may take up to 300% (as shown in Figure 5.4).

Remark. Our evaluations across various workload and infrastructure setups showed that JellyBean efficiently computes and deploys execution plans and significantly reduces the total serving cost of real ML workloads. We believe it is beneficial to leverage JellyBean for serving ML on heterogeneous infrastructures across a wide range of real-world applications.

5.6.3 Ablation Study

We leverage the `medium` setting and evaluate JellyBean by sweeping different knobs used during query optimization. We also demonstrate similar experiments on other setups in Appendix D.

Input throughput. To demonstrate the scalability of JellyBean and to supplement Figure 5.4,

Table 5.6: Costs of operators upon the `medium` setup.

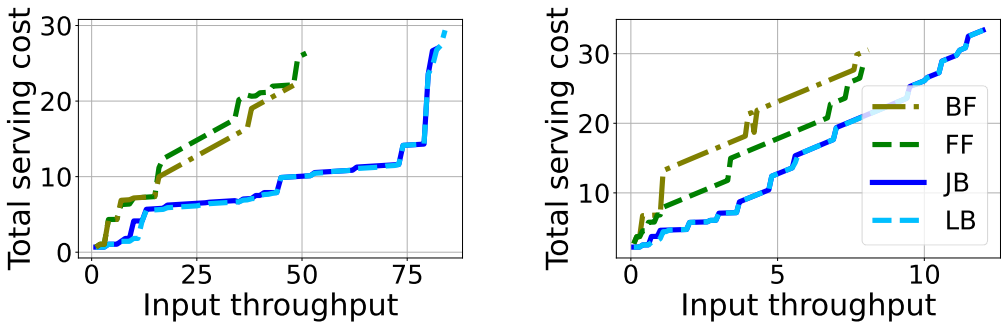
VQA Operator	Node	Orignial (ms)		QP exec.		QP network	
		P50	P90	P50	P90	P50	P90
ImgFeat	x2E	152.2	161.9	+2.5%	+4.1%	+1.2%	+1.3%
ImgFeat	x4E	76.6	82.3	+7.4%	+8.1%	+3.9%	+3.6%
ImgFeat	x16E	27.4	29.4	+11.7%	+19.4%	+6.1%	+6.0%
ASR	x8E	251.0	297.6	+2.3%	+1.0%	+0.4%	+0.4%
ASR	V100C	24.1	26.3	+0.8%	+0.8%	+0.6%	+0.5%
VQA	x48C	6.3	8.6	+9.7%	+12.6%	+4.8%	+6.9%

AICity Operator	Node	Orignial (ms)		QP exec.		QP network	
		P50	P90	P50	P90	P50	P90
ObjDet	x2E	1412	1455	+3.4%	+8.2%	+0.3%	+0.4%
ObjDet	x4E	721.0	732.0	+6.4%	+8.7%	+0.4%	+0.7%
ObjDet	x8E	451.7	468.3	+3.5%	+6.1%	+0.8%	+0.8%
ObjDet	V100H	19.7	20.6	+13.7%	+13.3%	+8.2%	+10%
ReID	V100C	8.4	8.7	+6.5%	+7.5%	+6.0%	+6.2%
ReID	V100C	8.5	8.8	+6.1%	+7.6%	+20%	+20%

we leverage a fixed target accuracy as in `medium` and demonstrate how the costs change when varying the input throughput. Figure 5.7 shows the results. We observe that JellyBean can keep up with increasing input throughput and is near optimal – in most situations, `JB` achieves the same total serving costs as `LB`. For `BF` and `FF`, no valid execution plans can be found beyond 51 rps and 8 fps (VQA and AICity, respectively).

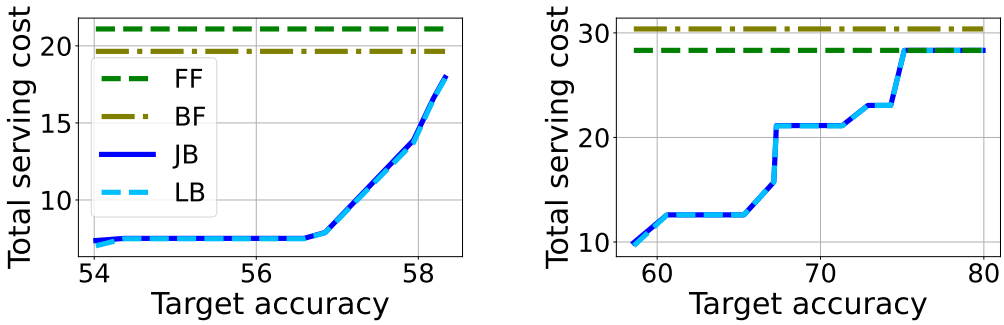
Target accuracy. To show that JellyBean provides viable accuracy-cost trade-offs, we fix the target throughput as in `medium` and demonstrate the total serving costs by varying the target accuracy. Figure 5.8 shows the results. `BF` and `FF` solve only for placement while using the most accurate models, and thus the costs are constant. For the scenarios we examined, JellyBean is near optimal across a range of accuracy targets. `JB` and `LB` eventually use the most accurate models, converging with `FF` for AICity.

Effect of model selection. To examine the model selection strategy used in JellyBean (Section 5.4.2), Table 5.7 illustrates an ablation study in which we substitute our model selection for either the most accurate models or a brute force selection. We also evaluate our model



(a) VQA (b) AICity

FIGURE 5.7: Total serving cost w.r.t. input throughput in JB.



(a) VQA (b) AICity

FIGURE 5.8: Total serving cost w.r.t. target accuracy in JB.

selection strategy for `PTc` and `SPc`. Results show that our proposed model selection is effective with our JellyBean processor as well as other ML runtimes.

5.6.4 Sensitivity Analysis

We further study the robustness and flexibility of JellyBean (`G4`) with the following sensitivity analysis experiments.

Effect of resource over-subscriptions. When there are more resources than needed, especially on the cloud, can JellyBean handle the workloads without wasting resources? Also, how do the costs change? We answer these questions by deploying the `small` workload on the `medium` infrastructure (Table 5.4). Figure 5.9 illustrates the results. We observe that, compared with using the `small` infrastructure, more resource availability will not significantly increase the serving cost for JellyBean with a fixed workload. However, `BF` and `FF`

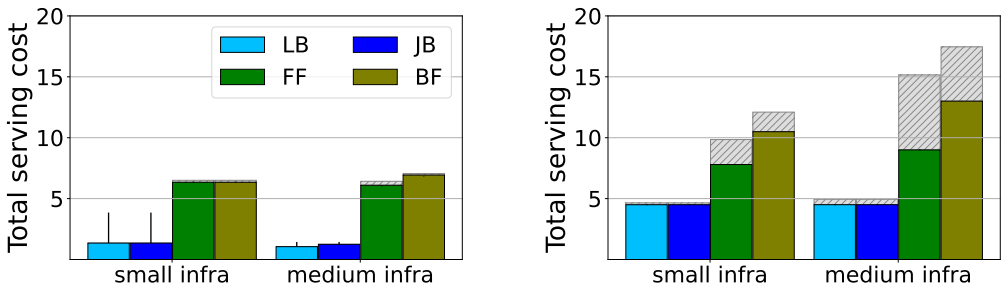
Table 5.7: Ablation analysis of model selection on the AICity dataset.

Model		medium			large		
Select.	Assign.	Comp	Net	QO	Comp	Net	QO
JB	JB	12.7	8.4	6.5ms	17.0	13.6	7.8ms
Most acc.	JB	14.3	14.0	1.2ms	17.5	19.1	1.3ms
Brute f.	JB	12.7	8.4	11.6ms	17.0	13.6	15.0ms
JB	PTc	5.3	27.2	N/A	7.3	37.4	N/A
JB	SPc	5.8	27.2	N/A	8.0	37.4	N/A
Most acc.	PTc	7.6	27.2	N/A	10.4	37.4	N/A
Most acc.	SPc	8.7	27.2	N/A	12.0	37.4	N/A

cannot guarantee cost efficiency in such a scenario. This is largely due to their sub-optimal worker assignment strategies which disregard resource availability. With JellyBean, users may use large cloud subscriptions without wasting resources.

Base unit network costs. We examine the effect of varying the network costs in a `medium` setup, which play a critical role in the total serving costs. Figure 5.10 showcases a change in cost from 0 to 1 (per GB). Note that on AICity, different lines are near linear after 0.12 and hence we show cropped results. Interestingly, for VQA, we found that the unit network costs actually have minor effects on the execution plans and the plan changes are subtle – this is due to a relatively higher compute cost on the cloud, so the computation is kept at the edge. Meanwhile, on AICity, we use blue dots to show where the plan changes, though the total serving cost is near linear. We present actual query plans in Figure 5.11 to show an example plan change when the network cost is reduced by 90% and compute is shifted to higher-tier workers.

Effect of infrastructure changes. We examine the flexibility of JellyBean when there are additional resource caps on the `medium` setup. Specifically, we placed a 10Mbps bandwidth constraint over all edge devices, mimicking real-world scenarios with limited networking. The JellyBean optimizer simply applies an additional constraint and limits the search space; there is no change on the processor and execution engine. Figure 5.12a illustrates the results and our findings. To reduce network costs, the JellyBean optimizer uses more compute



(a) VQA (b) AICity

FIGURE 5.9: Applying the `small` workload setup on the `medium` infrastructures. JellyBean uses the minimum available resource to achieve an optimal performance.

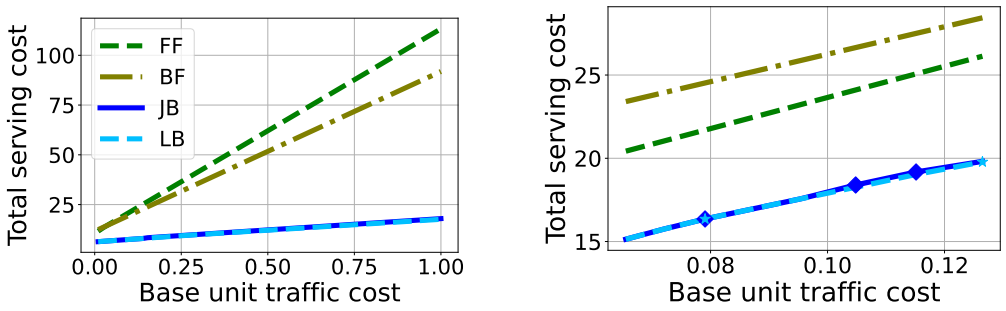
resources on the edge. The blue curve ends early since no viable solution can be found.

We change the number of workers allocated to different tiers, and observe how the total serving cost changes (Figure 5.12b). 'X' indicates unsolvable inputs. Since there are 9 total workers in the `medium` setting, we rank them according their cost and place those with higher costs on higher tiers (and vice versa). For instance, in the 5:4 case, the edge has 1x2, 1x4, 2x8 workers, and the cloud has 1x16, 1x48, 3xV100. Results show that JellyBean successfully finds good execution plans in all the settings; more cloud resources does increase the network costs.

Effect of profiling. In our previous experiments in Section 5.6.3, we leveraged the profiling of P75 percentiles as inputs to our QO and report actual runtime numbers; doing so gives extra room for the QO to find valid plans. In Figure 5.4, we also explore the estimated costs using P50 and P90 efficiency profiles on the error bars. We observe a small variance – the the actual runtime using P75 in most cases falls in the middle of estimates using P50 and P90. In some cases, the optimizer chooses different plans which leads to discontinuity of the costs. Overall, we observe that this has minor effects on our end-to-end solution.

5.7 Discussion

Obtaining diverse model choices. The user optionally provides a list of model choices for each operator in the workflow. Our current prototype depends on this provided model choices. However, in the future JellyBean can also enrich the choices using off-the-shelf



(a) VQA (b) AICity

FIGURE 5.10: The overall serving costs w.r.t. base unit traffic cost. Dots indicate when the execution plan changes.

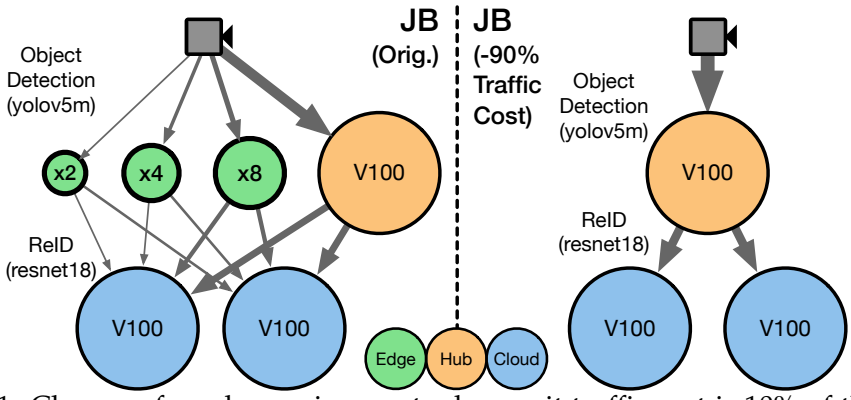
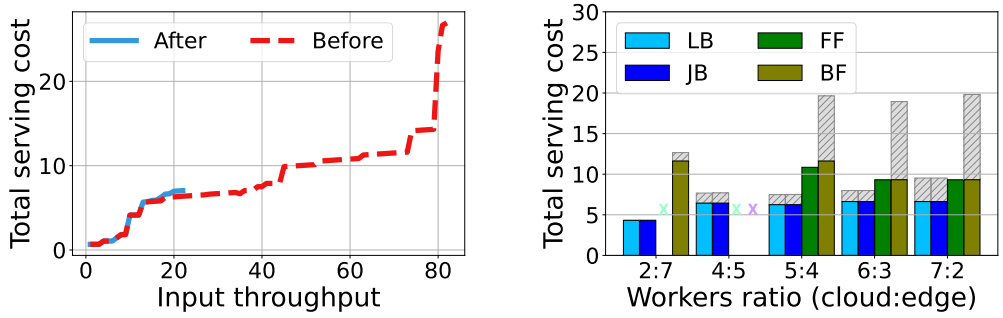


FIGURE 5.11: Change of worker assignment when unit traffic cost is 10% of the original traffic cost on `medium` setup for AICity.



(a) (b)

FIGURE 5.12: Sensitivity study on VQA for (a) limiting total outbound bandwidth and (b) changing the worker ratios on different tiers.

model quantization, pruning, and distillation tools. Several tools already exist today, and it is an active area of research in ML [100, 36, 213, 155, 243, 51] in order to democratize ML on weak edge devices. To integrate these tools into JellyBean, we can simply invoke them to derive cheaper models offline (similar to how we profile models for their accuracy profiles). We acknowledge that running these tools may require us to access the original training data and labels.

Limitations. As discussed in Section 5.4 and Section 5.5, we used a one-to-one mapping between the workers and operators. Using a one-to-multiple mapping to consolidate the operators may further improve the performance and can be an interesting further work to explore. Doing so may require automatic grouping of the operators. Nevertheless, we have shown in Section 5.6.3 that our processor already has low overhead.

JellyBean also assumes the heterogeneous infrastructures to have near constant input requests on the edge devices; this is true for the use cases discussed in Section 5.2 and in our experiments. Exploring use cases that do not fall into this category, such as security sensors or cluster telemetries which send only intermittent signals, can be an interesting future work. Besides, we used one-time profiling and fixed worker costs in our experiments; quickly adapting to changes in these aspects can also improve the usability of our system.

5.8 Related Work

Edge-cloud systems. Moving compute to the edge can reduce the networking cost and is used in video analytics to eliminate the need to transfer raw video streams. Chameleon [108] leverages temporal and spatial correlations to tune frame resolution, sampling rate, detector model configurations for an optimal resource-accuracy trade-off. In [281], a latency and energy consumption model is considered for choosing the configuration. Jain et al. [101] scale video analytics to large camera deployments using hand-crafted rules that leverage cross-camera correlation to improve cost efficiency and accuracy. Elf [306] applies a content-aware approach to offload smaller inference tasks in parallel to edge servers. These works considered a simple edge-cloud infrastructure and used workload-specific optimization tech-

niques. We support optimizing and running arbitrary ML workflows on a wide range of infrastructures, both of which are inputs to our optimizer.

ML inference systems. Serving machine learning inference has attracted great attention. TensorFlow Serving [197] is one of the first serving systems for production environments. Clipper [41] maximizes throughput under a user-specified latency service-level objective (SLO), model selection policies are also integrated to provide different cost-accuracy trade-offs. Nexus [250] automatically chooses the optimal batch size and the number of GPUs to use according to the request rate and latency SLO. Model DAGs are also considered in other works [42, 237, 275, 238, 87, 86]. JellyBean differs in two ways. First, we choose individual models based on input throughput and target accuracy for the entire ML workflow. Second, we target at deploying ML workflows on heterogeneous infrastructures, where prior works focused on either: a) homogeneous cloud datacenters or edge devices only, or b) heterogeneity within a single tier (i.e., datacenter).

Optimizing ML queries A number of works have been proposed in optimizing ML queries at either logical- or physical-level. Lu et al. [160] filter data that does not satisfy the query predicate by using probabilistic predicates. BlazeIt [116] optimizes aggregation and limit queries for videos. Yang et al. [295] exploit predicate correlations to build proxy models online to avoid exhaustive offline filter construction. Optimization at physical execution-level is addressed in some of the ML serving systems that support model DAGs. For instance, Llama [238] applies a greedy strategy that chooses cost-efficient worker configurations for video analytics pipelines. These works did not consider network cost, because these systems target pure datacenter deployment scenarios. JellyBean optimizes general ML workflows jointly at logical and physical levels for a heterogeneous infrastructure across edges and the cloud.

5.9 Summary

The rise of smart home devices and the Internet of Things opens up the opportunity for ML serving systems at the level of both the infrastructure and ML workflow to explore

new trade-offs between accuracy and performance. We build JellyBean, an ML serving to optimize ML workflows which takes into account the cost, availability, and performance of the increasingly tiered and heterogeneous infrastructures. JellyBean significantly reduces the total serving cost of visual question answering and vehicle tracking from the NVIDIA AI City Challenge compared with state-of-the-art solutions.

6. Lazarus: Resilient and Elastic Training of Mixture-of-Experts Models with Adaptive Expert Placement

In Chapter 5, we explore machine learning inference workflows and how we can use a system service to atomically deploy and serve them in a heterogeneous environment across edge and cloud with a cost-efficient execution plan. In this chapter, our focus shifts to another type of emerging workload and infrastructure: mixture-of-experts (MoE) models and spot instances. MoE models have increasingly been adopted to further scale large language models (LLMs). However, training or fine-tuning large MoE models still pose high resources requirements and result in staggering costs. Spot instances present an opportunity to significantly reduce training costs [268, 47, 9]. However, frequent preemptions of spot instances and the dynamic nature of MoE workloads make it challenging to utilize spot instances. We present our solution, Lazarus, a system that manages and optimizes MoE models training on spot instances, providing resiliency and elasticity to harness the cost efficiency of spot instances.

6.1 Introduction

To combat the prohibitive training cost of massive large language models (LLMs), the sparsely activated Mixture-of-Experts (MoE) models have increasingly been adopted by the community. Unlike dense layers activating all parameters for a given input, in the MoE architecture, each input is only forwarded to a subset of multiple parallel sub-modules (i.e., experts). The selective activation of parameters leads to sub-linear scaling of computation with model sizes. Still, training MoE models requires tremendous resources. For instance, it takes 50 days to train the 540B PaLM model on 6,144 TPUv4 chips [39] .

The likelihood and frequency of failures significantly increase as the scale and duration of training increase. Even a single failure is costly, as all GPUs are idling until failure is resolved and failed nodes are replaced. It is reported that failure rates can amount to 44% for LLM training [83] and slows the training progress by up to 43% [162]. In addition, most cloud providers offer preemptible (spot) instances that can be leveraged for training

LLMs with minimized monetary cost, as they offer cost savings of up to 90% compared to on-demand instances. Preemptions can happens as frequent as every 5~10 minutes in spot instance environment [268], which are essentially "failures".

Existing systems for LLM training with quick failure recovery can be categorized into two classes: in-memory checkpoint based or pipeline-parallelism based. The first line of works [286, 285] utilizes CPU memory of neighboring nodes to periodically checkpoint model states. However, they lack elasticity and have to wait for replacement nodes of the failed ones to recover from failure and continue training, which may not be available for hours to days until failed nodes are repaired [83]. Especially for training on spot instances, such new node availability cannot be taken for granted.

The second line of works builds resiliency into pipeline parallelism by taking advantage of its configurability in stages-nodes mapping [268, 103, 47]. They achieve both resiliency and elasticity, since they can continue training upon failures without requiring additional nodes. However, these approaches do not apply to MoE models, as the distributed training of MoE models include a different parallelism strategy: expert parallelism (EP) [138], where experts are distributed across multiple GPUs (nodes) and all-to-all communication is used to dispatch input tokens to GPUs with corresponding experts.

In this chapter, we present Lazarus, a system for resilient and elastic training of MoE models. Lazarus enables high-throughput training accompanied by a high probability of failure recovery without restarting from checkpoints. Upon failures, Lazarus quickly reconfigures the training job and fully utilizes all remaining GPUs (regardless of how many nodes fail).

Our insight is that adaptively adjusting the number of replicas (GPUs) assigned to each expert and their placement can improve resiliency against failures and performance due to imbalanced expert load distribution. Due to the dynamic nature of its architecture, MoE models suffer from dynamic and imbalanced workload [92, 300, 191]. Tokens are routed to experts based on the decisions of trainable gate networks. Some experts have more tokens routed to than others. In traditional EP, all experts are partitioned into equal-sized

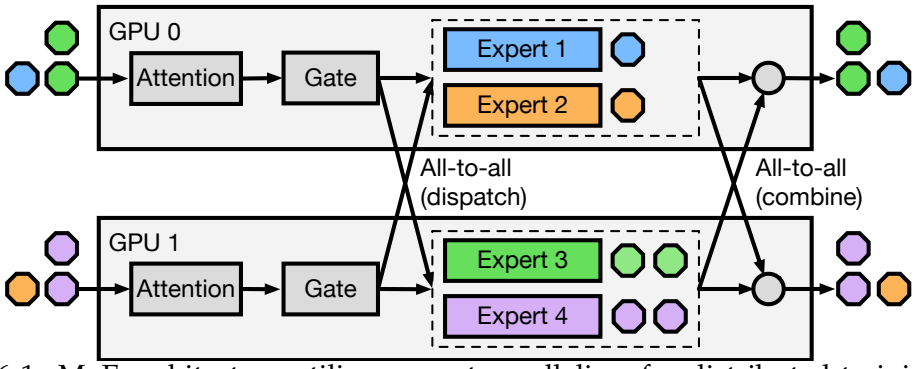
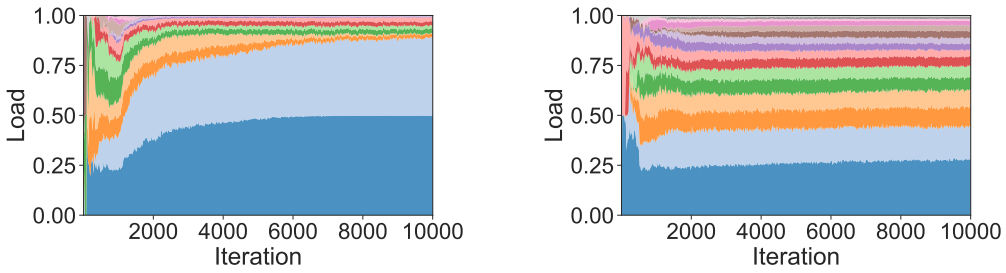


FIGURE 6.1: MoE architecture utilizes expert parallelism for distributed training, yet it suffers from imbalanced workload due to the dynamic nature of gate networks.

chunks and each chunk is assigned to the same number of GPUs. In contrast, Lazarus allocates more replicas to popular experts and flexibly assigns them using all available GPUs. Such flexible expert allocation not only results in performance boosts, but also leads to better training resiliency and elasticity. As long as a single replica for each expert remains available, training can continue to progress with all remaining nodes utilized; in traditional EP, only a multiple of EP size GPUs can be used, which can induce significant performance degradation even for minor failures.

There are three key challenges Lazarus must address. First, we need a expert allocation and placement algorithm that takes account of the imbalanced workload, to speed-up expert computation while maximizing the probability of successful recovery. Second, with our asymmetrical expert placements in the cluster, how do we efficiently dispatch tokens to GPUs with corresponding experts and balance their loads? Third, how do we quickly re-instantiate lost expert replicas and efficiently migrate the cluster to a new placement plan in response to failures?

To address these challenges, we propose a strategy for allocating expert replicas based on the load distribution, while maintaining a fault-tolerant threshold to guarantee failure recovery when a small number of nodes fail. We design a provably optimal algorithm for placing these replicas based on the idea of maximum rank overlap, to maximize the recovery probabilities under arbitrary node failures. We develop a CUDA kernel that dispatches to-



(a) Layer 1 (b) Layer 8

FIGURE 6.2: The expert loads on a 16-experts model (GPT-L in Section 6.6.1) The distribution varies during training and across layers.

kens in parallel with a flexible all-to-all that minimizes inter-GPU communication. During migration, Lazarus utilizes a greedy strategy to reduce state transfers for efficient reconfiguration.

We implement Lazarus in PyTorch. We evaluate Lazarus across MoE models of different scales with both controlled failures and spot instance traces. Our results show that Lazarus outperforms checkpointing-based DeepSpeed MoE [231], a widely adopted system for training MoE models by up to 2.3x under infrequent failures and 5.7x under a high failure frequency, while our evaluation on a real spot instance trace demonstrates a performance improvement to 3.4x.

In this chapter, we make the following contributions:

- To the best of our knowledge, Lazarus is the first system for resilient and elastic training of MoE models that enables both quick recovery from failures and full utilization of all available GPUs.
- We design a provably optimal algorithm for determining expert placement that maximizes recovery probability in response to uniformly random node failures.
- We implement and evaluate Lazarus with MoE models of different scales under a variety of scenarios.

6.2 Background and Motivation

6.2.1 MoE Models and Expert Parallelism

Mixture-of-Experts architecture has been recently applied to scale LLMs due to its high cost-efficiency, which replaces the dense feed-forward network (FFN) in a transformer block. MoE employs multiple parallel FFNs called experts. In each MoE layer, a trainable gate network routes each token to only the top-k experts (k is usually 1 or 2 [138, 55]). As experts are sparsely activated, MoE enables scaling model parameters without the increase of the per-token computational cost.

As the size of an MoE model is dominated by the weights of the experts, expert parallelism (EP) [138] has been proposed and has become the de facto approach to train large-scale MoE models. In expert parallel training, the experts of each layer are split into equal-sized chunks and allocated across multiple GPUs similar to tensor parallelism, while the input samples are distributed along batch dimension similar to data parallelism. The number of GPUs required to split the experts is called EP size and such a set of GPUs forms an EP group. For instance, in Figure 6.1, there are 4 experts and each GPU accommodates 2 experts, therefore it has a EP size of 2. EP can be used in conjunction with other types of parallelism like data and tensor parallelism.

As each GPU in an EP group only holds a subset of experts, all-to-all communication is used to dispatch the input tokens to the GPUs with corresponding experts that the gate network routes to. The computation of the experts are performed on the owning GPUs and the results are sent back to the original GPUs with a second all-to-all (combine).

The most distinctive feature of expert parallelism is the dynamic nature of gate networks. The distribution of tokens routed to each expert can be highly unbalanced depends on the input data. We plot the evolution of expert loads from a training trace [300]. We observe that the load of experts is highly skewed, with up to 87% tokens routed to 2 most popular experts. The load distribution also varies at different layers and training iterations.

The skewed expert loads in MoE training directly translates to imbalance in expert

computation. GPUs holding more popular experts takes much longer time to compute due to large amount of tokens dispatched to them, while other GPUs are idling. Previous works [92, 191, 300, 82] addresses this challenging by dynamically adjusting parallelism strategies on a cluster with a fixed number of GPUs. They do not apply in an elastic environment with changing device membership.

In addition to the problem of imbalanced workload, traditional EP also utilizes a multiple of EP size GPUs, which may leave some of GPUs idle upon a failure. The waste of GPUs only grows with increasing number of experts, as more GPUs are needed for a single EP group, i.e., larger EP size.

6.2.2 Fault-Tolerant and Elastic Training

A growing research effort has been made in resilient and elastic training in recent years, contributed to the fact that both the frequencies and costs of failures increase as the scale and duration of training increase. It is reported during the two months training of OPT 175B, around 100+ failures have been encountered [305], wasting over 178,000 GPU hours. The cost of even one failure is significant, as all the GPUs must wait idle until the failure is resolved and failed nodes are repaired, which could take hours to days depends on the nature of failures [83]. To minimize the GPU idling and the resulting economic loss, a training system must be designed with resiliency in terms of it can quickly recover from failures, and elasticity in terms that it can efficiently utilize currently available GPU resources to continue training. Such systems also enable one to leverage preemptible instances on public clouds to train LLMs with a significant cost savings [268, 47].

Existing solutions for fault-tolerant and elastic training can be divided into two categories: checkpointing based or pipeline parallelism based. Checkpointing based solutions periodically store model states to remote persistent storage, which has a significant overhead both in saving checkpoints and restarting. For instance, checkpointing the model states to remote storage for MT-NLG takes 42 minutes under a bandwidth of 20 Gbps [258].

Although in-memory based checkpointing [285, 286] has been proposed to reduce the

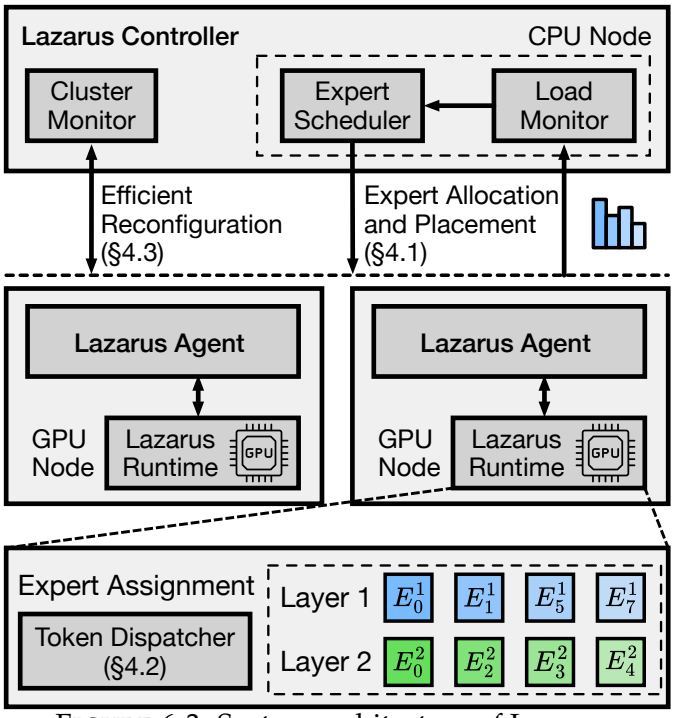


FIGURE 6.3: System architecture of Lazarus.

failure recovery overhead, they lack elasticity as they have to wait until replacements of failed nodes are available to resume training.

To support both elastic and fault tolerant training without the overhead of checkpointing and restarting, recent attempts [268, 103, 47] have been made in building resiliency into pipeline parallelism due to its configurability. However, these approaches fail to apply to MoE models due to the new computing paradigm introduced by expert parallelism.

In summary, existing systems for fault-tolerant and elastic training fail to adapt to MoE models. Lazarus targets MoE training, utilizing adaptive expert allocation and placement to address expert parallelism's inelastic nature while handling the imbalanced expert load distribution caused by the dynamic gate networks.

6.3 System Overview

Lazarus is a resilient and elastic system for training MoE models. Lazarus speeds-up training by adaptively allocating expert replicas based on the dynamic expert load distribution using all available GPUs, while our fault-tolerant expert placement strategy maximizes

Lazarus's recovery probability even under simultaneous failures of multiple nodes.

The architecture of Lazarus is shown in Figure 6.3. Lazarus consists of three main components: a centralized controller that manages a GPU cluster, an agent process on each GPU node that spins up worker processes with Lazarus runtime. The controller runs persistently on a (CPU-only) node and it communicates with each Lazarus agent, monitors the cluster and detects node failures and replenishment. A scheduler in the controller allocates expert replicas and computes a fault-tolerant placement plan (Section 6.4.1) for all GPU nodes. The placement is sent to each Lazarus agent to configure the workers. Based on the placement plan, Lazarus runtime fills up each layer with corresponding expert assigned to it. Unlike vanilla expert parallelism where all experts are equally replicated, Lazarus assigns more replicas and more GPUs to the heavily loaded experts. As the expert placements becomes asymmetric, Lazarus runtime also contains a CUDA kernel based dispatcher (Section 6.4.2) to efficiently dispatch tokens to GPUs with corresponding experts and balance their loads.

Upon detection of failures, the controller recomputes an expert placement plan using all remaining nodes and minimizes the number of replicas migrated. Once Lazarus runtime receives the new plan relayed by Lazarus agent, it dynamically reconfigures the parallelism setups and retrieves missing model states from other nodes (Section 6.4.3). To handle dynamics in workloads, Lazarus agent also periodically collects the expert load distribution (routing history of gate networks) from Lazarus runtime. The load distribution is communicated to the load monitor on the controller, which then rebalances the expert allocation and placement.

6.4 Design

6.4.1 Adaptive Expert Allocation and Placement

Lazarus considers that each GPU can hold a certain number of replicas limited by their GPU memory, similar to traditional EP. Lazarus speeds-up training by assigning more replicas to popular experts, corresponding to more computation resources. Note that we

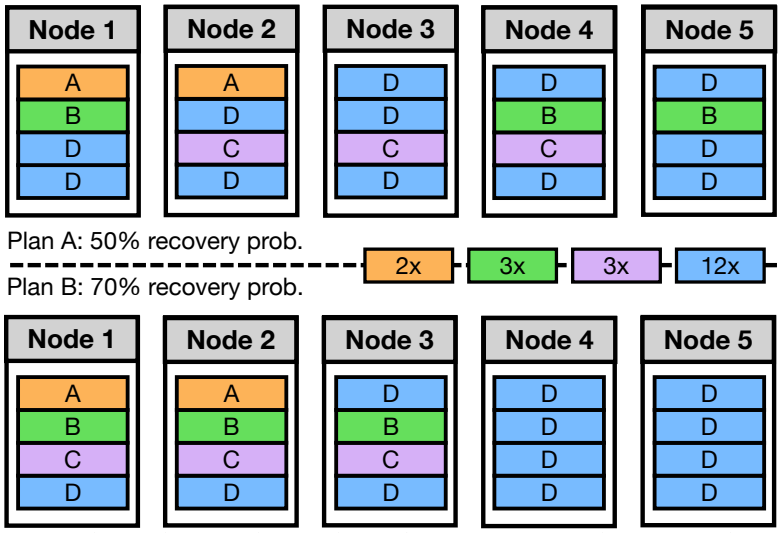


FIGURE 6.4: Fault resiliency depends on how expert replicas are placed.failures.

allow multiple replicas of the same expert assigned to a single GPU, which indicates more tokens (of the specific expert) can be processed by that GPU, comparing with assigning a single replica.

However, how the expert replicas are allocated also directly affects Lazarus's fault resiliency. For instance, if an expert is only assigned with a single replica, then as long as the GPU (node) holding that replica fails, Lazarus cannot recover due to the states of that expert is lost and have to restart from checkpoints. Moreover, even when the number of replicas allocated to each expert is fixed, the placement of these replicas determines the probability of failure recovery. For instance, if all replicas of an expert are all placed on GPUs in a single node, the loss of that node would lead to an unrecoverable failure. Hence, when allocating and placing expert replicas, Lazarus should not only consider the imbalanced workload to speed-up computation, but also take account of the impacts on fault tolerance.

Jointly satisfying these two goals is quite challenging due to two reasons. The first reason is that the search space for both expert allocation and replica placement is exponentially large, making it infeasible to enumerate over all possible plans. The second reason is that there is an inherent trade-off between the two goals. On one hand, a balanced allocation of

replicas over all experts minimizes the probability of unrecoverable failures (combined with an optimal placement plan); however, it degenerates to traditional EP and defeats the goal of addressing expert load imbalance. On the other hand, as more replicas are allocated for popular experts, the probability of failure recovery becomes lower for less popular experts with fewer replicas.

We divide the problem into two phases and separately consider allocation and placement. In the first phase, we design an expert allocation strategy that balances between the workload's expert distribution and fault tolerance. In the second phase, we design an expert placement algorithm which is theoretically optimal, maximizing the recovery probability given a fixed expert allocation. In this way, our allocation and placement plan strikes a balance between the two goals.

Expert allocation. For ease of presentation, we show how expert replicas are allocated and placed on each node. If a node has multiple GPUs, Lazarus simply distributes the assigned replicas among all GPUs on that node, as we consider failures at the node level. We denote the number of nodes as N, the number of experts as E, the number of replicas each node can hold as c, the total number of tokens routed to expert e as t_e, the number of replicas assigned for expert e as r_e. To speed-up computation, we want the ratio of replicas assigned to each expert match the ratio of tokens routed to that expert, namely $\frac{r_e}{\sum_{e'} r_{e'}} \approx \frac{t_e}{\sum_{e'} t_{e'}}$. Furthermore, for better fault tolerance, we define a fault-tolerant threshold f, where Lazarus guarantees a 100% probability of failure recovery as long as less than f nodes fail simultaneously. Hence, each expert is assigned at least f replicas.

Assume that the experts are sorted by number of routed tokens (t_e) in an ascending order. We iteratively compute the number of replicas r_e assigned to each expert e as follows:

$$r_e = \max\{\lfloor \frac{t_e}{\sum_{e'=e}^{E} t_{e'}} \cdot (N \cdot c - \sum_{e'=1}^{e-1} r_{e'}) \rfloor, f\} \tag{6.1}$$

Our assignment strategy ensures that $\sum_e r_e = N \cdot c, r_e \geqslant f, r_e \geqslant r_{e-1}$. $\frac{r_e}{\sum_{e'} r_{e'}} \approx \frac{t_e}{\sum_{e'} t_{e'}}$ is also satisfied in most cases for training speed-up under the imbalanced workload. As $r_e \geqslant f$,

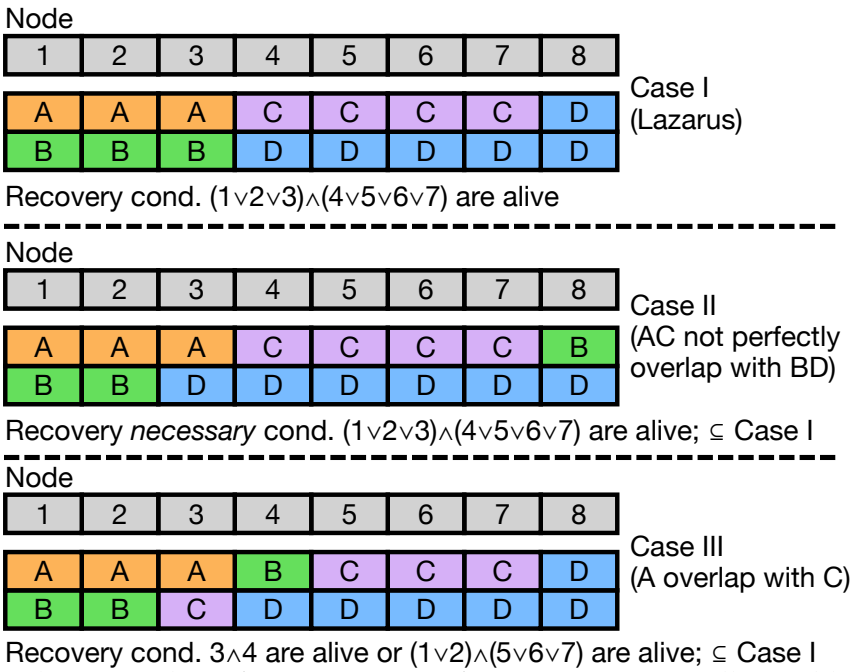


FIGURE 6.5: The optimality of Lazarus comes from catergorizing all possible placement plans based on whether representatives of each group are perfectly separated.

Lazarus guarantees recovery under failures of a small number ($< f$) of nodes.

Expert placement. However, when the number of failed nodes $\geqslant f$, the probability of failure recovery differs between different placement plans. Figure 6.4 shows an example of 4 experts and 5 nodes. With the same replica allocation of 4 experts and 4 replicas slots per node, placement plan A and B differs in recovery probability under 3 node failures. In plan A, the probability of recovery is $\frac{5}{10}$, as recovery is possible only if the alive nodes are $(1,2),(1,3),(1,4),(2,4),(2,5)$, while there are 10 possible cases. In plan B, however, the probability of recovery is much higher at $\frac{7}{10}$.

Placement solution for an easier case. We first consider a simpler case where $E \leqslant c$, which we can easily derive an optimal placement strategy inspired by the previous example. The strategy is for the first $\min(r_1, N)$ nodes we place the first (least popular) expert, for the first $\min(r_2, N)$ nodes we place the second expert, and so on. For the vacant positions, we uniformly place the experts that still have replicas left. In this way, denote the set of nodes that have the e-th expert as S_e. This strategy satisfies $S_1 \subset S_2 \cdots \subset S_E$. Thus the

recover probability is equal with the probability of the first expert belonging to an alive node (i.e., any of the first r_1 nodes is alive). Furthermore, the first expert belonging to an alive node is a necessary condition of failure recovery. Thus for any placement plan, the recovery probability is upper bounded by the probability of any of the first r_1 nodes is alive. Since there are only r_1 replicas for the first expert, it can span across at most r_1 different nodes. Therefore, in the case of $E \leqslant c$, this placement strategy achieves the upper bound of the recovery probability for all placement plans, guaranteeing its optimality.

The above strategy relies on two principles: (1) scatter less popular experts (with fewer replicas) across different nodes; (2) maximize the nodes overlapped between the experts. Letting each expert span across more nodes is always helpful, as the probability of recovering a single expert only depends on how many nodes hold this expert, regardless of the concrete set of nodes placing it. On the other hand, maximizing the overlap between experts benefits recovery by relaxing the conditions on node liveness. Take the first and second expert as example, if we overlap all the replicas of first experts with the second expert's replicas on the same nodes, the two experts' states will be lost only when all of the first expert's replicas are lost. However, if some of the first expert's replicas are not overlapped with the second expert's, the two experts cannot be recovered when either all of the first's replicas are lost or all of the second's are lost.

Placement solution for the more difficult case. When $E > c$, the optimal strategy becomes more complicated. The previous introduced maximum overlap principle cannot be directly applied due to the infeasibility of overlapping all experts when $E > c$. To address this issue, we partition the experts and nodes both into $\lceil \frac{E}{c} \rceil$ groups. We also modify the second principle into maximizing the overlap of experts in each group. Furthermore, we constrain the expert partitions to be consecutive, i.e., $1, \cdots, c$ are in the first group, $c+1, \cdots, 2c$ forms the second group, and so on. For the nodes, we divide the first $\min\{N, \sum_{i=1}^{\lceil \frac{E}{c} \rceil} r_{c*(i-1)+1}\}$ nodes into $\lceil \frac{E}{c} \rceil$ groups. The first group has r_1 nodes, the second has r_{c+1} nodes, and so on, while the last group has $\min\{N - \sum_{i=1}^{\lceil \frac{E}{c} \rceil - 1} r_{c*(i-1)+1}, r_{c*(\lceil \frac{E}{c} \rceil - 1)+1}\}$ nodes. For group i in the

first $\lceil \frac{E}{c} \rceil - 1$ groups, each node contains one replica of expert $c*(i-1)+1, \ldots, c*i$. For the last group, each node contains one replica of expert $c*(\lceil \frac{E}{c} \rceil - 1)+1, \ldots, E$. For the vacant slots, we uniformly place the experts that still have replicas left to place. Our strategy satisfies $S_{c*(i-1)+1} \subset S_{c*(i-1)+2} \cdots \subset S_{c*i}$ for different i, which intuitively maximizes the node overlapping of experts in each group. The recovery of the experts in the $i-$th group hence only requires one node in $S_{c*(i-1)+1}$ to be alive, where we define the expert $c*(i-1)+1$ as the representative of group i. The complete recovery is equivalent to that one replica of each group's representative still remains. Our maximum rank overlap (MRO) placement plan maximizes recovery probability under uniformly random node failures for any given replica number r. Concretely, we have Theorem 1. The proof can be found in Appendix F.

Theorem 1. *For any MRO plan T and R, given the number of replicas r_e for each expert e, T maximizes the recovery probability $\mathsf{Pr}(\bigcup_{a \in A} Col_a = [E])$, where $[E]$ is the set of experts, Col_a is the set of replicas assigned to node a, A is a uniformly sampled set of R nodes that remain alive.*

Here we offer some core ideas of proving the optimality of our solution. We use Figure 6.5 as a motivating example, where $E = 4, c = 2$. Let us first consider the recovery of the first $c+1$ experts. All possible placement plans of this example can be partitioned into two types: (1) The first expert and $(c+1)$-th expert overlaps. (2) The first expert and the $(c+1)$-th expert do not overlap. Case III in Figure 6.5 represents type (1), Case I and Case II belong to type (2). For plans that satisfy type (1), from the figure we can see that their recovery condition for the first $c+1$ experts is stricter (comparing Case III to Case I). This shows the sub-optimality of overlapping the first expert and the $(c+1)$-th expert. For plans that satisfy type (2), since the first expert and $(c+1)$-th expert do not overlap, the recovery probability of the first $c+1$ experts is upper bounded by the probability of recovering both the first and $(c+1)$-th expert, due to necessary condition. Furthermore, the probability of recovering both the first and $(c+1)$-th expert is identical for different type (2) plans since the first and $(c+1)$-th expert do not overlap so the placement position does

not influence recovery of these two experts. Notably, for Case I (Lazarus), its probability of recovering the first $c+1$ experts equals the probability of recovering both the first and $(c+1)$-th expert since they are the representatives of the two groups. Therefore Case I achieves optimal for recovering the first $c+1$ experts.

Now instead of only considering the first $c+1$ experts, we consider all E experts. For Lazarus, its recover probability for E experts is identical with recovering the first $c+1$ experts, since the representatives of the two groups remains the same (in Case I, recovering (A, B, C) is equivalent with recovering (A, B, C, D)). Due to necessary condition, for any placement plan, its recover probability on E experts is upper bounded by the probability of recovering the first $c+1$ experts. Recall the optimality of Lazarus on recovering the first $c+1$ experts, thus its also achieves optimal recovery for E experts.

From the previous case we observe that, for $\forall E' > c$, for the placement plan that maximizes the recovery of all E' experts, the placement of the least popular c experts exactly align the optimal placement plan for the first c experts, and the remaining decision becomes how to place the left $E' - c$ experts. This makes it possible to obtain the optimal solution for $E' > 2c$ by recursively placing the least popular c experts satisfying maximum overlap and reducing the problem to a sub-problem of expert number $E' - c$.

We note that the expert load distribution can be different across layers, hence we compute an expert replica allocation and placement plan independently for each layer. As the load distribution also shifts during training according to the workload, Lazarus also periodically rebalances the expert allocation and updates the placement plan.

Now, we have developed the strategy to assign expert replicas to each node (GPUs). Next, we explore under such asymmetric replica placements, how Lazarus efficiently dispatches tokens to GPUs with replicas of routed experts.

6.4.2 Flexible Token Dispatcher

In traditional expert parallelism, each token can be simply dispatched to the GPU that owns the corresponding expert, as there is only a single replica for each expert within a

Algorithm 2: Token dispatch algorithm.

Input : N: Number of GPUs; i: Current GPU rank; h: Activation of input tokens to the MoE block; $R_{e,j}$: Number of replicas for expert e assigned to rank j; $T_{e,j}$: Number of tokens routed to expert e at rank j;

Output: h': Shuffled inputs for all-to-all dispatch; s_j: Number of tokens to dispatch to rank j

for $e \leftarrow 0$ **to** E *in parallel* **do**
  $r_e \leftarrow \sum_j R_{e,j}$ // total #replicas for expert e
  $t_e \leftarrow \sum_j T_{e,j}$ // total #tokens routed to expert e
  $p_e \leftarrow t_e / r_e$ // #tokens each replica should handle
  for $j \leftarrow 0$ **to** N *in parallel* **do**
    $P_{e,j} \leftarrow c_e R_{e,j}$ // #tokens rank j can process
    $P_{e,j} \leftarrow P_{e,j} - \min(P_{e,j}, T_{e,j})$
    // rank j's local tokens are prioritized
  $D_{e,i} \leftarrow c_e R_{e,i} - P_{e,i}$ // locally processed #tokens
  for $j \leftarrow 0$ **to** N, $j \neq i$ *in parallel* **do**
    $D_{e,j} \leftarrow (T_{e,i} - D_{e,i}) \frac{P_{e,j}}{\sum_{k \neq j} P_{e,k}}$
    // distribute remaining tokens to other ranks
for $j \leftarrow 0$ **to** N *in parallel* **do**
  $s_j \leftarrow \sum_{e'} D_{e',j}$ // #tokens dispatched to rank j
  for $e \leftarrow 0$ **to** E *in parallel* **do**
    $start \leftarrow \sum_{0..j-1} s_{j'} + \sum_{0..e-1} D_{e',j}$
    $end \leftarrow \sum_{0..j-1} s_{j'} + \sum_{0..e} D_{e',j}$
    $h'[start..end] \leftarrow (\sum_{j'=0}^{j-1} D_{e,j'})$-th to $(\sum_{j'=0}^{j} D_{e,j'})$-th tokens in h that routed to e
return h', s

particular EP group. Concretely, an all-to-all is performed with all ranks (GPUs) in the EP group sending and receiving the same number of tokens, which is dynamically set to the maximum number routed to a single expert to prevent token dropping, while unused slots are padded [231, 92].

With Lazarus's adaptive expert placement, there are varying numbers of replicas assigned for each expert on different sets of GPUs. Therefore, each rank must decides which rank with the routed expert's replica to dispatch a token to. The fact that multiple replicas can be assigned to the same rank (indicating more tokens should be dispatch to it), combined with the difference in expert routing on different ranks, a challenge emerges — how can we efficiently dispatch the tokens to all GPUs with the routed experts while balancing

the load? If tokens are poorly dispatched, some ranks could receive significantly more tokens than others, hence defeating the purpose of our adaptive expert allocation. Moreover, the padded all-to-all is no longer viable in our case where a token can be dispatched to any rank (instead of within a EP group), as padding would dominate the communication.

To address these issues, we design a flexible token dispatcher that efficiently dispatches each token to a particular GPU and balances the number of tokens routed to each GPU. With the dispatch schedule computed, Lazarus performs a flexible all-to-all without padding. Algorithm 2 shows the workflow of the token dispatcher, which is implemented in a CUDA kernel to process all experts and target ranks in parallel. The basic idea behind Algorithm 2 is that each replica of an expert should compute around the same number of tokens, and each rank should utilize its local processing "capacity" before dispatching remaining tokens to other ranks.

Before computing the dispatch schedule, an all-gather is first performed to collect how many tokens are routed to each expert from all ranks, i.e., $T_{e,j}$. $T_{e,j}$ is collected so that the token dispatcher can better balance the load to each GPU based on the expert routing distribution of all tokens from all ranks, instead of using only locally computed tokens. In addition, since collective communication operations require synchronization of all participant ranks, $T_{e,j}$ is also necessary in computing how many tokens a rank should receive from each of the other ranks. Since only E integers are collected from each rank, this extra all-gather imposes negligible overhead, as demonstrated in Section 6.6.5.1. The number of replicas allocated to each GPU $R_{e,j}$ from the placement plan is also passed to the token dispatcher.

After $T_{e,j}$ are collected, each rank i independently computes how many tokens it dispatches to each of all N ranks, for all E experts. First, for each expert e, the number of tokens each replica should process is computed in line 4 by evenly distributing all t_e tokens routed to e onto all r_e replicas. The processing capacity of each rank j can then be computed by multiplying p_e with the number of replicas of e that j is assigned (line 6). This capacity will be prioritized towards tokens computed locally on j. After the remaining capacities of

all ranks are computed, rank i dispatches the remaining $(T_{e,i} - D_{e,i})$ tokens that are beyond i's local processing capacity. The number of tokens $D_{e,j}$ to dispatch to each rank for e is calculated based on their residual capacities (line 10).

Since the all-to-all collective operates on a continuous buffer, the token dispatcher has to reshuffle the input activations h to the MoE block, so that tokens routed to the same expert and dispatched to the same rank are grouped together. The total tokens s_j to dispatch to rank j across all experts is computed in line 12. In line 13-16, these tokens are sorted by their routed experts and placed consecutively in h'. The reshuffled activations h' are then used in the dispatch all-to-all collective, with s_j tokens sent to each rank j. The token dispatcher also computes how many tokens to receive from each rank j in the all-to-all in a similar fashion.

At this point, Lazarus has the ability adaptively assign expert replicas and dynamically dispatches tokens among replicas of routed experts. Next, we discuss how Lazarus efficiently migrates to a new configuration upon failures.

6.4.3 Efficient Reconfiguration

As discussed in Section 6.4.1, if at least a single replica of each expert still remains upon failures, Lazarus can recover without restarting from checkpoints. However, the remaining expert replicas' distribution could deviate from the desired allocation computed for the workload, and their placement may be prone to subsequent failures. Therefore, Lazarus must reallocate expert replicas and efficiently migrate to a new placement plan. Such migration is also required when Lazarus rebalances the expert allocation and when new nodes join.

The ordering of nodes in the placement plan is not enforced in the placement algorithm, as long each node in the plan maps to a physical node in the cluster. However, when migrating from an old placement plan, such a mapping becomes relevant. It directly determines how many experts' states a node needs to retrieve from other nodes, as only newly assigned ones not in the old placement plan have to be fetched. To reduce the number of replicas

to shuffle during migration, hence the communication, Lazarus applies a greedy algorithm that iteratively maps a physical node to a node in the new placement plan that the number of newly assigned experts is minimized.

After the node mapping is determined, Lazarus schedules the transfers of expert states. Each node fetches missing states for the newly assigned experts from other nodes that own them. If multiple nodes require the states of the same expert, Lazarus distributes their state transfers among all owning nodes, to minimize the overall migration time.

6.5 Implementation

Lazarus is implemented in 4K LoC in Python and 500 LoC in CUDA, building on top of PyTorch [95] (v2.3) and using components from DeepSpeed [233] (v0.13).

Lazarus controller and agents. We implement the controller and agents using Python's asynchronous framework. New agents register with the controller, using a TCP socket for communication. The controller maintains a global view of node availability, where agents periodically sending heartbeats for it to detect failures. Upon failures or scaling up with newly arrived nodes, the controller computes an updated expert placement plan, which is sent to each agent and relayed to the worker process that uses Lazarus runtime. The agents also periodically collect expert routing history from each worker and send it to the controller for expert rebalancing.

Lazarus runtime. Based on the controller's configuration, our runtime sets up NCCL [189] communication groups for expert and non-expert gradients all-reduce, as well as all-to-all in expert computation. We implement data parallelism and expert parallelism with our adaptive expert placement; however, Lazarus can be extended to combine with pipeline parallelism using techniques like Oobleck [103], which are orthogonal and complementary to ours. Upon failures, enqueued NCCL operations time out and the model states are not updated on the failed step, while a new configuration is received from the agent via a listener thread. Batched NCCL send/recv primitives are used to transfer states during migration. For scaling up and rebalancing, Lazarus performs reconfiguration lazily, only

Table 6.1: Configurations of models used in the evaluation.

	GPT-S	GPT-M	GPT-L
# Layers	12	12	12
Feature dim.	768	1024	1024
# Experts	8	12	16
# Params	521M	1.3B	1.7B

after when the current training step is finished.

6.6 Evaluation

6.6.1 Setups

Testbed. We have five servers in our testbed, each with 2 NVIDIA RTX 3090 GPUs and a 100 Gbps Mellanox ConnectX-5 NIC connected to a single 100 Gbps Mellanox SN2100 switch. Due to limited resources, we treat each GPU as a separate node to emulate a cluster of 10 GPUs. To store checkpoints, we deploy a NFS server on a separate machine, which is connected to the GPU servers via 10 Gbps NICs.

Baselines. As there is no existing system to support resilient and elastic training of MoE models, we compare Lazarus against a checkpoint-based baseline using DeepSpeed MoE (DS) [231], which is a state-of-the-art implementation of MoE training system with both system-side and model design-side optimization. To evaluate Lazarus's adaptive expert placement algorithm and flexible token dispatcher, we also build a fault tolerant baseline based on DeepSpeed MoE, utilizing efficient reconfiguration module from Lazarus runtime. We denote this baseline as DS(FT). Similar to Lazarus, if a complete replica of all experts still exists upon failures, it reconfigures the workers (reassigns EP groups) and retrieves required model (expert) states from owning nodes.

Workloads. Based on the widely used GPT-2 architecture, we adopt three MoE models of varying sizes and number of experts, listed in Table 6.1. We use a per-GPU batch size of 4 and a sequence length of 1024 following GPT-2's setup [55]. For all evaluation, we use Wikitext-2 dataset [171], top-1 gate and FP16 precision for training.

For reproducibility, we use the routing history trace from SmartMoE [300] artifact to

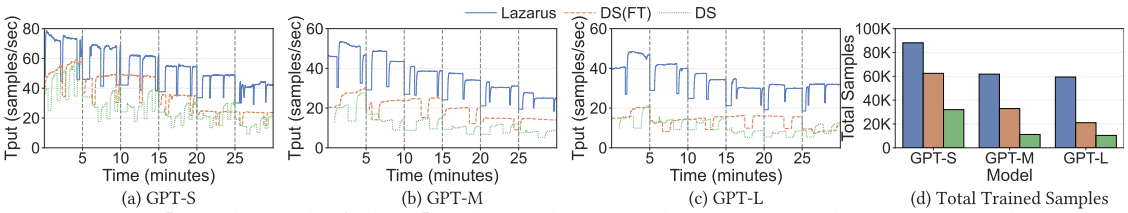


FIGURE 6.6: [Single node failure]: Throughput and total trained samples with a single node fails every 5 minutes.

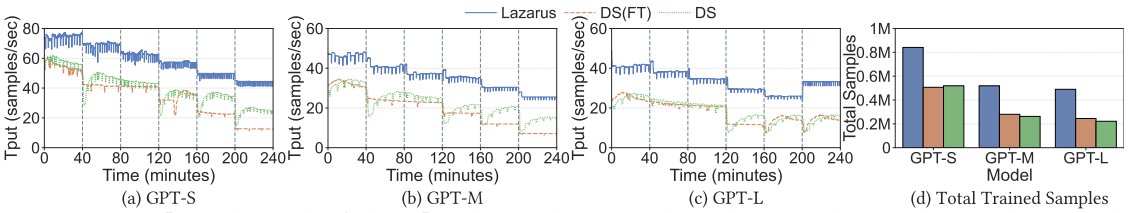


FIGURE 6.7: [Single node failure]: Throughput and total trained samples with a single node fails every 40 minutes.

emulate gate networks' routing decisions. We use the loads of top experts at each layer to construct a routing trace for each of the models we evaluate. We set the number of expert replica slots for each GPU to 6, which is the upper limit based on available GPU memory. With DS's traditional expert parallelism, GPT-M can fully utilize all slots, while GPT-S and GPT-L can only use 4, as the multiple of slots per GPU and EP size must equal to the number of experts. GPT-S and GPT-M can utilize an EP size of 2, hence DS and DS(FT) fully utilize all 10 nodes in the cluster, while with 16 experts and an EP size of 4, they can only utilize 8 nodes on GPT-L. We set the checkpoint interval to every 50 steps for DS and every 250 steps for DS(FT), unless mentioned otherwise. We set the minimal replicas per expert (f) to 2 for Lazarus so that recovery is guaranteed under common single node failure scenarios. Lazarus rebalances expert replica allocation every 200 steps.

6.6.2 Controlled Single Node Failures

We first evaluate the performance in a more common case where a single node fails at a time. We consider both a high failure frequency and a low frequency scenarios, where we randomly choose a node to fail every 5 or 40 minutes, until only half of the nodes remained. The same set of nodes are selected to fail in each run for fair comparison. The results are

shown in Figure 6.6 and Figure 6.7. The throughput is smoothed over a short time window for visibility. The fluctuation in Lazarus's throughput is caused by the reconfiguration after node failures and the periodical rebalance of expert allocations, while the fluctuation in DS and DS(FT) is contributed to checkpointing, restarting and reconfiguration (only for DS(FT)). To reduce the overhead of checkpointing for DS and DS(FT) in the low failure frequency (40 minutes) setting, we increase their checkpoint intervals by 4x to per 200 steps and 1000 steps, receptively. We also note that using such low checkpoint frequency would prevent DS from making any effective progress under high failure frequency (5 minutes).

From Figure 6.6 with a failure frequency of 5 minutes, we observe that over the 30 minutes duration of training, Lazarus finished a total of 2926 and 1996 steps, trained 2.8x and 5.7x samples on GPT-S and GPT-L, compared with DS. The performance gains significantly increase on GPT-L, as the checkpointing and restarting overhead grows with model sizes. Moreover, as in the GPT-L setting (EP size is 4), 4 nodes are required to hold a complete replica of all experts for DS and DS(FT), they can only utilize either 4 or 8 nodes, while they can utilize all 10 nodes at the start for GPT-S and GPT-M.

Lazarus also outperforms DS(FT) by 1.4x and 2.8x on GPT-S and GPT-L. On the smaller GPT-S, there are a large number of replicas for each expert (5 replicas initially), hence DS(FT) can recover in each failure. However, as the number of experts and EP size increases on GPT-L, DS(FT) have to restart from checkpoints after failures of both EP groups.

When the failure is infrequent as shown in Figure 6.7, the performance difference between Lazarus and DS decreases as the overhead of checkpointing and restarting decrease. Still, Lazarus outperforms DS by 1.6x and 2.3x on GPT-S and GPT-L. As the overhead of DS decreases, DS and DS(FT) have similar performance in this case.

We also observe that Lazarus's throughput tends to monotonically decrease as the number of nodes decreases, as Lazarus can fully utilize all remaining nodes for training. While for DS and DS(FT), the throughput experiences steep drops since they can only utilize a multiple of EP size nodes. We note that the throughput of Lazarus increases in the last 40

minutes in Figure 6.7. This is because Lazarus no longer enforces a minimal of 2 replicas for each expert for fault tolerance, as there are not enough slots with 5 nodes left.

Lazarus still outperforms DS by a great margin, even when both of them fully utilize all 10. For instance, for GPT-M, during the first 40 minutes in Figure 6.7 when no node fails, Lazarus has a throughput of 45 samples/sec during effective computation (factoring out checkpoint and rebalance overheads), while DS only reaches 34 samples/sec.

From Figure 6.6, we also observe that compared to DS, DS(FT) sometimes has higher throughput during effective computation. For instance, for GPT-M, DS(FT) outperforms DS by 1.6x during the 5~10 minutes window, when they all fully utilize the remaining 8 nodes. This is mainly caused by the highly imbalanced expert loads during the early periods of training, while DS(FT) progresses much faster without the overhead of checkpoint and restarting. When we increase the checkpoint intervals by 4x for both baselines in Figure 6.7, together with the lower failure frequency, such divergence disappears. Instead, for GPT-S and GPT-M, DS(FT) is slower than DS during the last 80 minutes. In these two cases, DS(FT) always resumes training by reconfiguring currently used nodes that are still alive. It does not use previously dropped nodes (due to exceeding EP size of 2), while DS attempts to utilize all nodes it can when restarting.

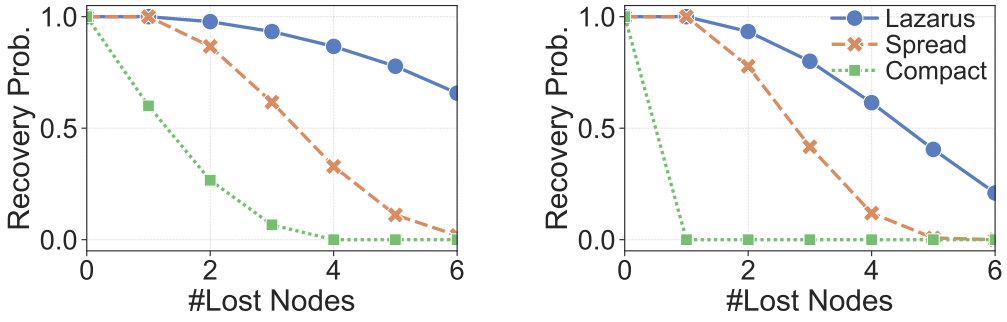
Overall, checkpointing and restarting overhead becomes increasingly significant with larger models and higher failure frequency. Comparing with DS(FT) which shares Lazarus's efficient reconfiguration runtime, Lazarus's adaptive expert placement improves both training throughput and resiliency.

6.6.3 Controlled Multi Node Failures

Next, we study how well does Lazarus handle simultaneous failures of multiple nodes. Whether Lazarus can recover from such failures depends on both the expert allocation (i.e., how many replicas assigned to each expert) and expert placement, as well as which concrete set of nodes fail. The allocation and placement changes as the expert load distribution varies over the duration of training, and it is also different for different layers. Hence, we evaluate

Table 6.2: [Multi-node failures]: Recovery overhead of Lazarus under multiple node failures on sampled cases.

	GPT-S		GPT-L	
	step 200	step 4000	step 200	step 4000
# Lost nodes	2	3	4	5
Reconfig time (s)	21.3	34.1	18.2	19.7
# Experts transfer	11	52	160	55
Transfer time (s)	2.3	3.0	7.6	7.8

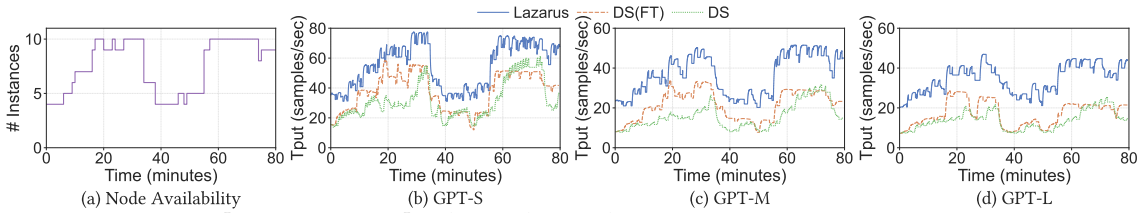


(a) GPT-S (step 4000) (b) GPT-L (step 200)

FIGURE 6.8: [Multi-node failures]: Recovery probabilities using different expert placement strategies.

Lazarus's system overhead of recovery by sampling several cases for GPT-S and GPT-L at different training steps, while we evaluate Lazarus's placement algorithm by computing the recovery probability for a model at a given training step. The recovery probability can be computed by enumerating all possible combinations of failed nodes, as the how experts are allocated and placed only depends on the expert load at the particular step.

The recovery overhead for sampled cases is shown in Table 6.2, where 2 to 5 nodes are



(a) Node Availability (b) GPT-S (c) GPT-M (d) GPT-L

FIGURE 6.9: [Spot instance]: Throughput changes in spot instance environment.

selected to fail at training step 200 and 4000. We report the total number of experts replicas that need to be transferred between nodes and the time spent on the state transfers. The weights and optimizer states of each is 63MB for GPT-S and 112MB for GPT-L. We find that the overhead of state transfers is negligible. This low overhead is mainly contributed by the fact that required states can be fetched from other nodes instead of the much slower remote storage, and Lazarus balances the point to point send/recv operations among all owning ranks of an expert's states. We also report the total reconfiguration time, from failure occurrence to training resumption, where state transfers only constitutes a small portion. Throughout our entire evaluation, we find that each reconfiguration event takes 20~40 seconds. It takes 10~20 seconds for enqueued NCCL kernels to timeout and 5~15 seconds for reconfigure NCCL's communication groups. We also observe that the placement plan's computation take less than 100ms.

To demonstrate the effectiveness of Lazarus's fault-tolerant expert placement algorithms, we compare it with two baselines: a spread placement strategy which distributes each expert's replicas across different nodes in a round-robin fashion, and a compact strategy that packs an expert's replicas on minimal number of nodes. The recovery probabilities with respect to the number of nodes failed are illustrated in Figure 6.8. We find that Lazarus's placement algorithm greatly outperforms both baselines. For instance, for GPT-L at step 200, Lazarus has a 41% recovery probability with 4 node failures, compared to 12% of spread placement. We also observe that on the smaller GPT-S when expert loads are relatively more balanced at later step 4000, compact placement achieves limited recovery capability with 1 or 2 node failures. However, it completely fails to recover in any failure scenario on the larger GPT-L with 16 experts.

6.6.4 Spot Instance Trace

We also borrow real spot instances node availability trace from Bamboo [268] to evaluate Lazarus under both failures and scaling-up. The trace includes both preemption events and node additions. We replay a representative 80 minutes segment of the availability trace

collected on AWS EC2 P3 instances. As the original trace is collected on a 32 nodes cluster, we cap the maximum number of nodes to 10 in our testbed setup. To handle rare cases where recovery is not possible due to too many nodes failing at the same time, we also apply periodic checkpointing for Lazarus. We set the checkpoint interval to every 250 steps, same as DS(FT), for fair comparison. For node additions events, all compared methods wait for 2 minutes to accumulate sufficient nodes before scaling-up, to avoid frequent reconfigurtion or restarting. The results are shown in Figure 6.9.

Over the 80 minutes duration, Lazarus trained 2.3x and 3.4x samples on GPT-S and GPT-L, compared with DS. Lazarus outperforms DS(FT) by 1.2x and 1.8x on GPT-S and GPT-L. We also note that Lazarus's throughput changes proportionally to the number of nodes available, as Lazarus wastes no node, while DS and DS(FT) are limited by EP sizes.

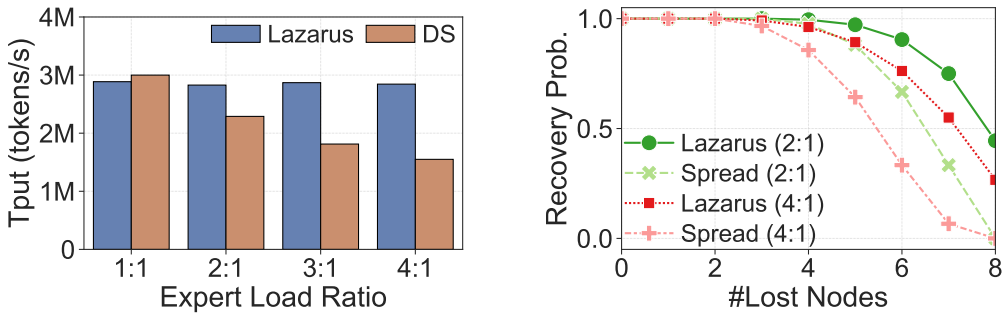
Due to the overhead of checkpointing and restarting, DS trained 51% and 48% less samples than DS(FT). DS(FT) can always recovery from failures for GPT-S and GPT-M, as it evenly allocates up to 5 replicas to all experts at a cost of reduced throughput. For GPT-L, however, when there is less than 8 nodes, DS(FT) cannot utilize more than 4 nodes for redundancy. It has to restart from checkpoint each time, leading to 3~5 minutes of lost progress.

We observe that only in a single preemption event when 4 nodes are lost at 34 minutes, Lazarus has to restart from checkpoint. Note that in the original trace, only a maximum of 19% nodes failed at a time.

6.6.5 Ablation Study

6.6.5.1 Impacts of Expert Load Imbalance

To study how the expert load imbalance in workloads affects both Lazarus's performance and fault resiliency, we build a single MoE layer with 8 experts and a feature dimension of 1024. We construct workloads with different expert load ratios. We show the layer forward throughput in Figure 6.10a. Here, a load ratio of 4:1 indicates that 4x more tokens is routed to one of the expert than if all experts are evenly routed to.



(a) Throughput (b) Recovery Probability

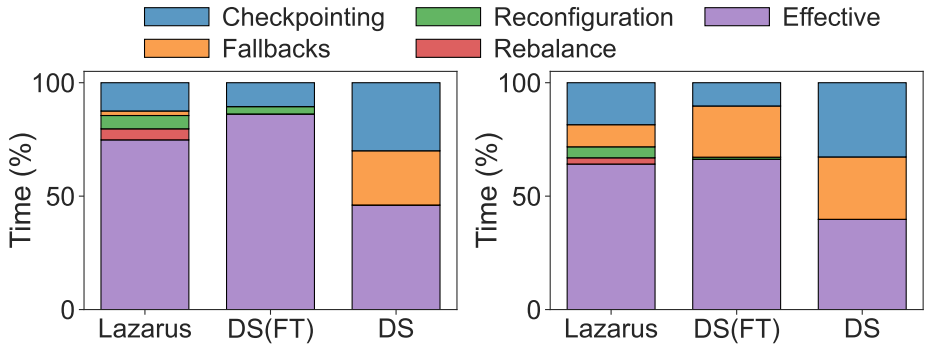
FIGURE 6.10: [Ablation Study]: Single layer throughput and recovery probabilities under different expert load ratios.

We observe that Lazarus's throughput remains constant as the load ratio changes, contributing by Lazarus's adaptive expert allocation based on expert load distribution. DS's throughput, however, dramatically decreases as the workload becomes more skewed. When the workload is perfectly balanced (1:1), Lazarus suffers a small overhead due to its flexible token dispatcher.

We also evaluate the effectiveness of Lazarus's expert placement algorithm in fault tolerance as the load distribution changes. Figure 6.10b shows the recovery probability of Lazarus with varying number of failed nodes on 2:1 and 4:1 load ratios, compared with the spread placement strategy. We observe that the recovery probability decreases with more imbalanced workload, as less popular experts are assigned less replicas. Still, Lazarus's placement algorithm is much more effective than spread placement, while our previous evaluation demonstrates the increased throughput is worth the effort of skewed expert allocation.

6.6.5.2 Running Time Breakdown

We breakdown the running time on the spot instance trace from Section 6.6.4 in Figure 6.11. Both Lazarus and DS(FT) has much more time spent in effective computation, benefiting from efficient reconfiguration module in Lazarus runtime, while over half of the time is spent on checkpointing and restarting (fallbacks) for DS. The reconfiguration and



(a) GPT-MoE-S (b) GPT-MoE-L

FIGURE 6.11: [Ablation Study]: Running time breakdown of GPT-S and GPT-L on the spot instance trace.

rebalance overhead of Lazarus is much smaller than restarting, accepting for less than 10%. We also find that DS(FT) can recover in all cases on GPT-S, yet it suffers 27% restarting overhead on GPT-L. Despite similar effective time, Lazarus outperforms DS(FT) by 1.8x in terms of total trained samples, contributed by our adaptive expert allocation and flexible token dispatcher.

6.7 Related Work

MoE training systems Extensive studies have focused on optimizing MoE training. A series of works [231, 251, 107, 140, 154, 192] optimize the all-to-all communication performance. Another line of works design different MoE algorithms and architectures [231, 192, 142, 313, 317, 318, 37]. Various system optimizations have been proposed to deal with the imbalanced workload. For example, Tutel [92] and SmartMoE [300] propose dynamic parallelism switching; FasterMoE [82] and FlexMoE [191] also utilize the idea of expert replication. However, these works all focus on speeding up training on a fixed-sized cluster, while Lazarus considers an elastic environment where resiliency and quick reconfiguration is crucial. Many of these optimizations can also be integrated to Lazarus.

Fault-tolerant and elastic training. Early efforts in elastic training focus on small models trained with pure data parallelism. TorchElastic [273] restarts a job upon node membership changes. Elastically allocating resources among multiple jobs have also been explored

in [224, 311, 91, 75, 141]. However, they do not work for modern LLMs which are frequently well beyond a single GPU's memory capacity. To enable efficient training of large models, in-memory checkpointing has been proposed in [285, 286], which lack elasticity and require replacement nodes to resume training. In particular, Gemini [286] designs a strategy for placing checkpoints in CPU memory across machines to maximize recovery probability. However, it assumes each GPU's checkpoint has the same number of replicas, hence does not apply to our expert placement problem, where different experts have different number of replicas. Systems supporting both resilient and elastic training of LLMs [268, 103, 47] are all based on pipeline parallelism, utilizing its flexibility in stage-device mapping. These works are orthogonal and complementary to Lazarus, where we target expert parallelism introduced in MoE.

6.8 Summary

This chapter presents Lazarus, the first system for resilient and elastic distributed training of Mixture-of-Experts (MoE) models. Lazarus adaptively allocates replicas based on the expert routing distribution of the workload to speed-up training. With a proven optimal expert placement strategy, Lazarus maximizes the probability of failure recovery. Upon failures, Lazarus efficiently migrates to a new expert placement plan with all remaining GPUs fully utilized. Our results show that Lazarus outperforms state-of-the-art checkpointing based MoE training systems by up to 5.7x under frequent node failures and 3.4x on a real spot instance trace. We will open source Lazarus.

6.9 Acknowledgement

The Lazarus project is in collaboration with Wenjie Qu. Wenjie contributed to the algorithm designs while I focused on the system part.

7. Conclusion

The growing scale and complexity of today's distributed workloads have driven application developers and infrastructure operators to increasingly demand cost efficiency and manageability, in addition to performance. Existing systems struggle to meet these goals, contributed by the detachment of workloads to the infrastructure, and the new deployment options introduced by new types of workloads. In this dissertation, we present our vision and a comprehensive, four-part approach to achieve these goals. I argue that decoupling the implementation of communication primitives and the control of deployment strategies from distributed applications can improve their performance, cost efficiency, and manageability.

To improve performance and manageability, on the lower-level datacenter communication side, we decouple the implementations of common communication primitives from libraries linked to user applications into managed systems services. In Chapter 3, we present mRPC, which decouples RPC marshalling and policy enforcement into the mRPC system service, providing policy flexibility and availability without the overheads from sidecars. We introduce MCCS in Chapter 4, which decouples the implementation of various collective communication algorithms into a service provided by the cloud, enabling topology-aware and cross-application optimization of collective strategies, while adding more management features like quality of service and network-aware traffic engineering.

To improve performance and cost efficiency, on the higher-level deployment side, we present systems that decouple the control of the workload deployment from applications. In Chapter 5, we present JellyBean, a system that serves machine learning workflows on heterogeneous infrastructure across edge and cloud, optimizing worker assignment and model selection to reduce serving costs. In Chapter 6, we implement Lazarus, a system that enables resilient and elastic training of mixture-of-experts (MoE) models on spot instances, taking advantage of their cost efficiency.

Collectively, these systems optimize the deployment of distributed workloads, improving performance, increasing cost efficiency and achieving better manageability. These techniques also lay the foundation towards a more flexible and resource efficient distributed

infrastructure in the future.

7.1 Future Directions

In this dissertation, we present systems that *independently* optimize the two factors: (1) lower-level communication and (2) higher-level application deployment. Looking ahead, I believe cross-layer optimization that *jointly* considering these two factors will further push the boundaries for performance, cost efficiency and manageability. In the following section, we will explore two potential future directions towards this goal.

Co-optimization of collective strategies and parallelism configurations in ML training. Today's ML training is increasingly bottlenecked by communication as model scales, which accounts for 30% to 95% of the overall training time [27, 283]. MCCS can improve training performance through optimizing the collective communication strategies and improving the performance of collectives. However, MCCS imposes no control on the communication paradigm in the training process, which is determined by how the model is parallelized on the cluster. Different parallelism techniques have been developed, and they have different communication patterns. For example, data parallelism uses AllReduce operations to synchronize gradients [143], while pipeline parallelism uses point-to-point send and receive operations to exchange activations between pipeline stages [188]. As large language models (LLMs) have scaled to hundreds of billions or even trillions of parameters [289, 156], they are often trained with a combination of multiple types of parallelisms. Tuning these parallelism configurations is shown to deliver significant improvements in training performance. As both the lower-level collective strategies and higher-level parallelism configurations greatly impact training performance, a system service that manages both would not only enforce how a collective is implemented, but also which collectives are issued, opening up further opportunities for performance optimization and reducing training costs.

Co-optimization of communication services and job schedulers in multi-tenant datacenters. As mRPC and MCCS only manage the data plane in a datacenter, they have no control on which sets of nodes a workload is deployed on. MCCS only attempts to optimize

the collective strategy given the sets of participant nodes. However, how the workloads are placed also greatly impact the communication performance [230]. For instance, modern datacenters are often oversubscribed, if a communication-intensive workload is scattered across nodes under multiple racks, it could leads to significant reduced inter-host significant. A future research avenue is to jointly optimize the job scheduler's workload placement and the strategies used by the communication services, further improving performance.

Appendix A. Evaluating mRPC with Full gRPC-style Marshalling

As gRPC uses `protobuf` [221] for encoding and HTTP/2 as the payload carrier, it has a memory copying and HTTP/2 framing cost. On the other hand, mRPC is agnostic to the marshalling format. Although mRPC's default marshalling is zero-copy and is generally faster than gRPC-style marshalling, our main goal is to show that we can eliminate the redundant (un)marshalling steps while enabling network policies and observability for RPC traffic.

To isolate the performance benefits of using zero-copy marshalling and reducing the number of (un)marshalling steps, we evaluate mRPC with full gRPC-style marshalling (protobuf + HTTP/2). We implement an mRPC variant that applies encoding (decoding) code generated by the `protobuf` compiler and HTTP/2 framing for inter-host mRPC service communication.

We conduct the same large RPC goodput experiment in Section 3.7.1 on TCP transport. The results are presented in Figure A.1. The error bars show the 95% confidence interval, but they are too small to be visible. We find that mRPC achieves performance comparable to gRPC after switching to using protobuf + HTTP/2. With full gRPC marshalling, mRPC still performs 2.6× and 3.7× as fast as gRPC + Envoy in terms of goodput and goodput per core. This is because mRPC reduces the number of (un)marshalling steps. The small RPC rate and scalability of mRPC with gRPC marshalling is also shown in Figure A.2, where the error bars show the 95% confidence interval. Since encoding small RPCs with protobuf is relatively fast, the trend to the rate and scalability is similar to Figure 3.5a.

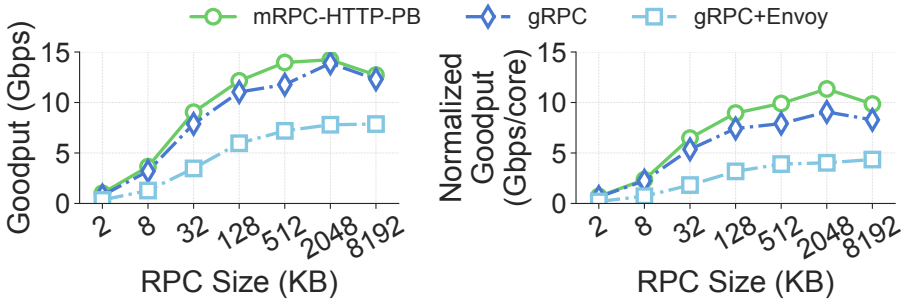


FIGURE A.1: Microbenchmark [Large RPC bandwidth]: Comparison of large RPC bandwidth where we use HTTP/2 and protobuf (PB) marshalling for mRPC.

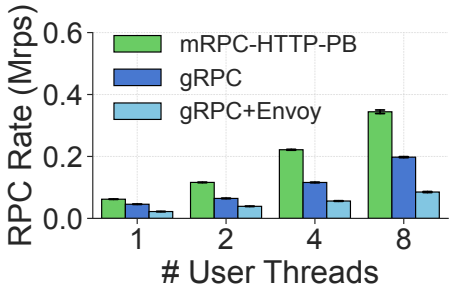


FIGURE A.2: Microbenchmark [RPC rate and scalability]: Comparison of small RPC rate and CPU scalability where we use HTTP/2 and PB marshalling for mRPC.

Appendix B. Extended Evaluation for DeathStarBench in mRPC

We report the P99 latency of DeathStarBench in Figure B.1, comparing gRPC with Envoy and mRPC. A null policy is applied for mRPC. The result is similar to the comparison of median latency in Section 3.7.4. mRPC speeds up gRPC+Envoy by 2.1× in terms of end-to-end P99 tail latency.

We also evaluate gRPC without proxy and mRPC without any policy enforced. Figure B.2 and Figure B.3 show the results for mean latency and P99 tail latency. We observe that mRPC speeds up gRPC by 1.7× and 1.6×, in terms of mean latency and P99 tail latency. Communication costs are substantial in the DeathStarBench applications, and thus reducing the communication latency can improve end-to-end application performance. This is consistent with the original DeathStarBench paper's observation [62].

We further compare the memory usage of gRPC and mRPC. The peak memory consumption of gRPC and mRPC in DeathStarBench applications is illustrated in Figure B.4. For mRPC, we report the user application side memory usage, which also includes all the memory pages shared with the mRPC service. We observe that mRPC does not incur notable memory overhead compared to gRPC. On the other hand, we find a small and constant memory footprint of mRPC service across all machines at around 9 MB.

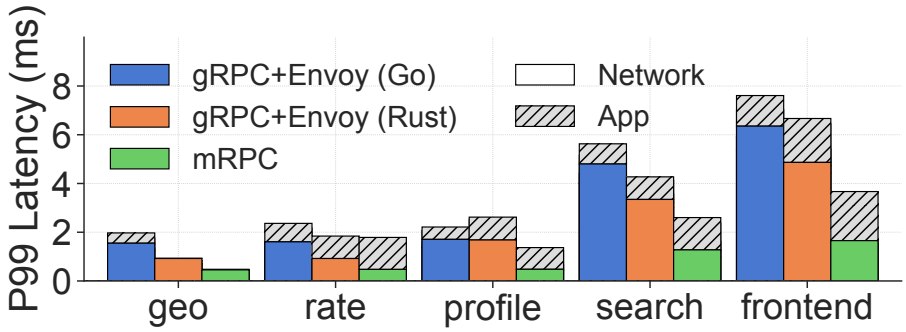


FIGURE B.1: DeathStarBench: P99 latency of in-app processing and network processing of microservices, respectively. gRPC with Envoy and mRPC are compared.

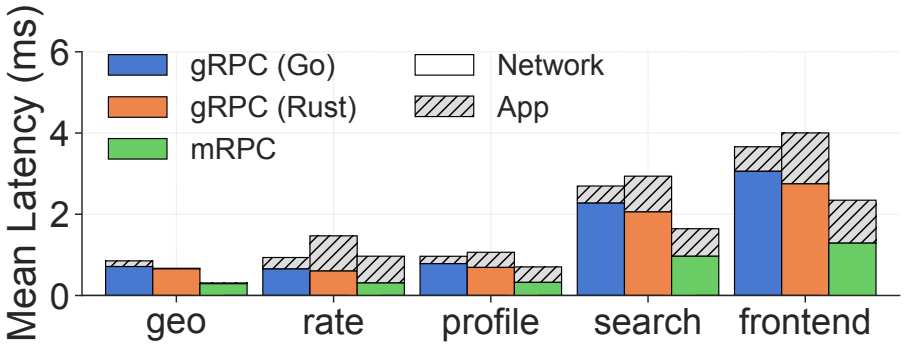


FIGURE B.2: DeathStarBench: Mean latency of gRPC without proxy and mRPC.

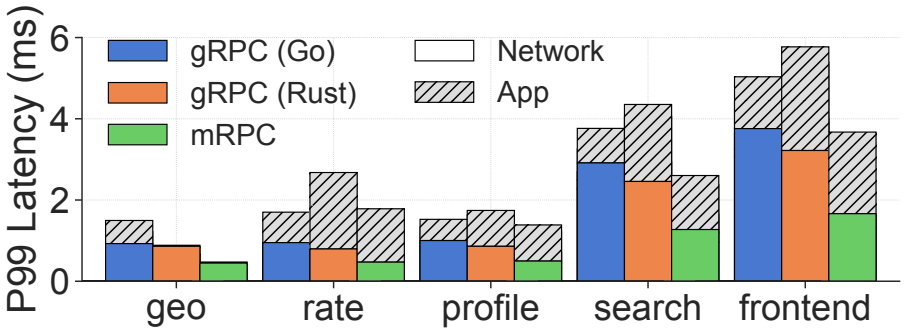


FIGURE B.3: DeathStarBench: P99 latency of in-app processing and network processing of microservices, respectively. gRPC without proxy and mRPC are compared.

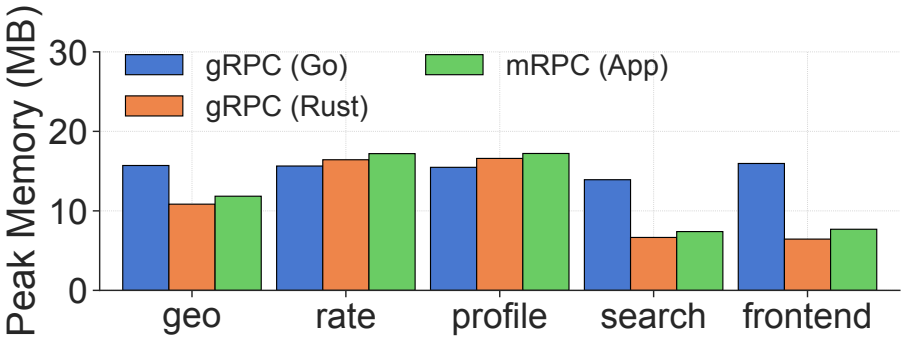
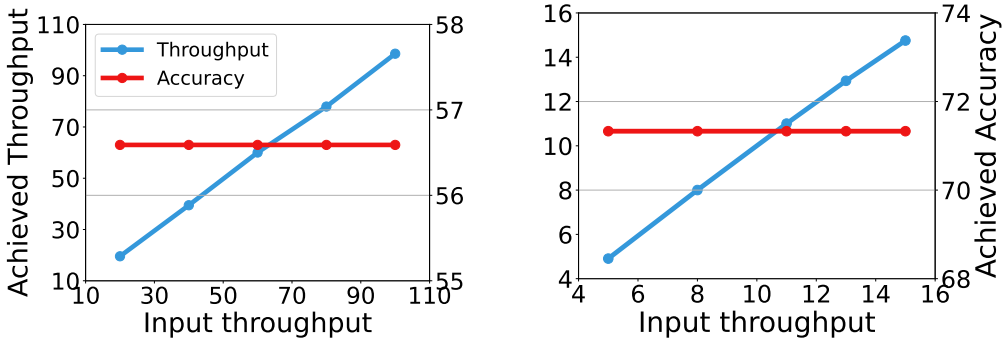


FIGURE B.4: DeathStarBench: Peak memory usages of different services. gRPC without proxy and mRPC are compared.

Appendix C. Extended System Evaluation of JellyBean

We illustrate the actual throughput and achieved accuracy with varying input throughput on the `large` setting in Figure C.1. Again, we find that JellyBean's actual throughput mostly reaches the target throughput. In some cases, the achieved throughput is slightly lower than the target (<2% lower). This might be contributed by the performance variation of the vCPUs on IBM cloud at different time-of-day, as we observed.

Here we also report the runtime variance on the `xlarge` setup. The standard deviation of the serving cost across 5 runs is 0.122% for AICity, and 0.028% for VQA.



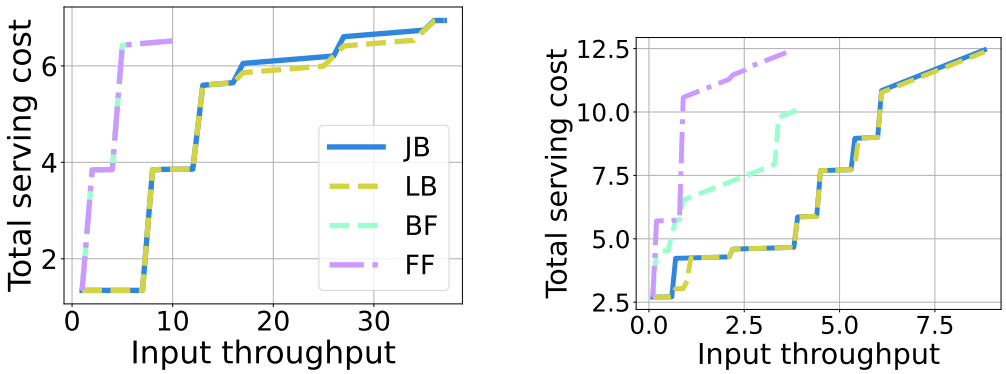
(a) VQA (b) AICity

FIGURE C.1: Achieved throughput and accuracy given input throughput on the `large` setup.

Appendix D. Extended Ablation Study of JellyBean

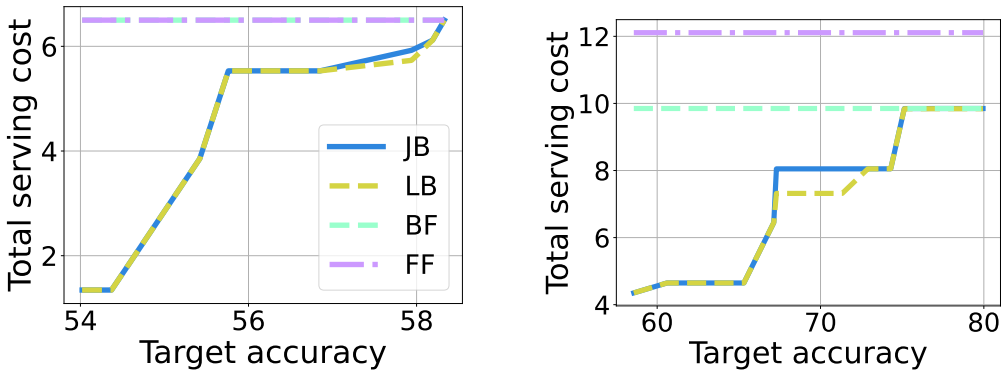
Here we demonstrate how the serving costs change with respect to input throughput and target accuracy for JellyBean and the compared baselines in `small` and `large` setups. The results on the `small` setup are shown in Figure D.1 and D.2, while the results on the `large` setup are shown in Figure D.3 and D.4. We do not report the serving costs for `LB` on the `large` setup, as they could not find the solution in a reasonable amount of time. From the figures, we observe that `JB` consistently outperforms `BF` and `FF`, and achieves the same or similar serving costs compared with `LB` on the `small` setup.

We also showcase the effect of model selection on the VQA dataset in Table D.1. The results again demonstrate the effectiveness of `JB`'s model selection strategy, as well as its effectiveness with other ML runtimes.



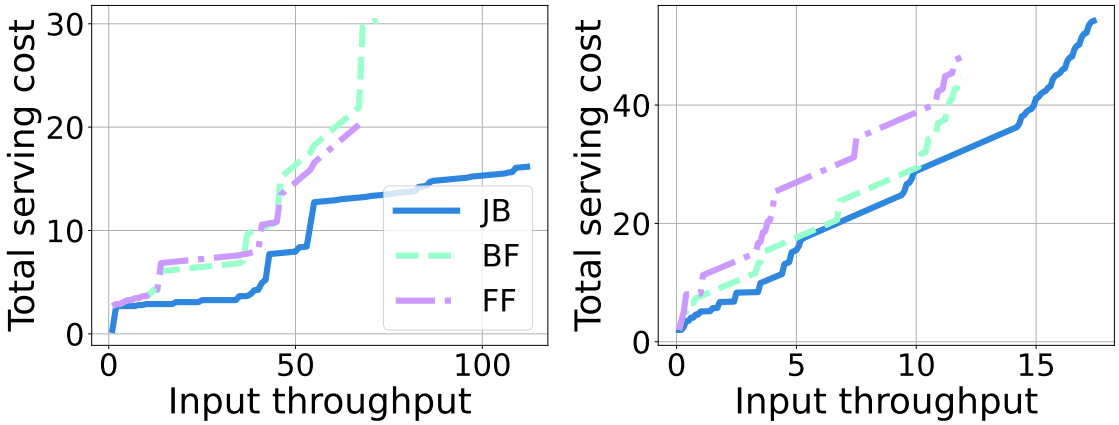
(a) VQA (b) AICity

FIGURE D.1: Total serving cost w.r.t. input throughput in `JB` on the `small` setup.



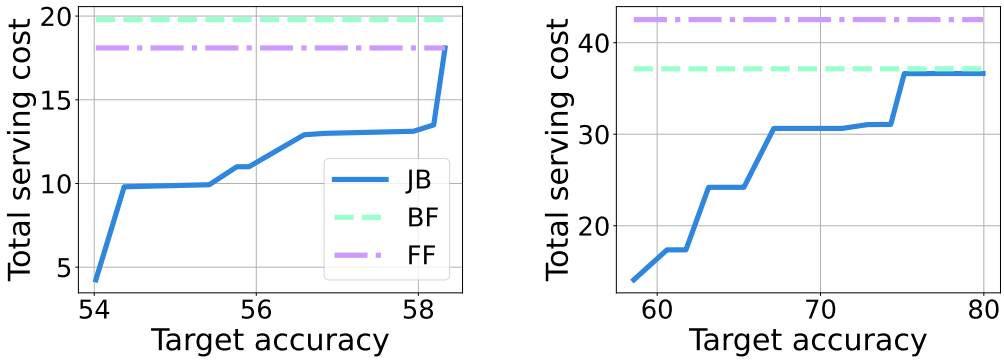
(a) VQA (b) AICity

FIGURE D.2: Total serving cost w.r.t. target accuracy in JB on the `small` setup.



(a) VQA (b) AICity

FIGURE D.3: Total serving cost w.r.t. input throughput in JB on the `large` setup.



(a) VQA (b) AICity

FIGURE D.4: Total serving cost w.r.t. target accuracy in JB on the `large` setup.

Table D.1: Ablation analysis of model selection on the VQA dataset.

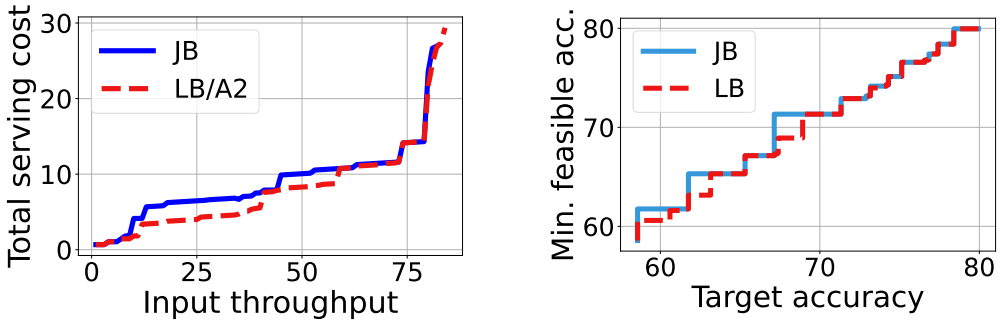
Model		`medium`			`large`		
Select.	Assign.	Comp	Net	QO	Comp	Net	QO
JB	JB	6.2	1.3	20.8 ms	11.3	1.6	29.3ms
Most acc.	JB	9.1	8.8	2.3ms	12.1	6.0	3.1ms
Brute f.	JB	6.2	1.3	21.3ms	11.3	1.6	30.0ms
JB	PTc	4.0	12.2	N/A	6.0	18.4	N/A
JB	SPc	15.8	12.2	N/A	23.7	18.4	N/A
Most acc.	PTc	5.8	12.2	N/A	8.8	18.4	N/A
Most acc.	SPc	19.3	12.2	N/A	29.0	18.4	N/A

Appendix E. Discussion on the Performance of JellyBean

We employ greedy approaches in the JellyBean query optimizer for both model selection and worker assignment. In this section, we further analyze: 1) why JellyBean often provides execution plans that are competitive with the lower bound in practical configurations, and 2) properties of configurations for which JellyBean may generate worse execution plans.

E.1 Why JellyBean performs well in practice

For most cases in Section Section 5.6, JB has a total serving cost close to LB, even though JB utilizes a greedy strategy. The small performance gap is due in part to the workflow properties and infrastructure configurations. On the infrastructure side, lower infrastructure tiers generally have fewer, less-capable workers than higher tiers; additionally, lower tiers have a reduced communication cost compared to higher tiers due to their proximity to the data sources. With Assumption A2, our worker assignment algorithm starts to assign workers from lower tiers considering both compute and communication costs. On the workflow side, the output accuracy of ML operators generally increases monotonically with respect to the improvement of the input quality; our beam-search leverages this property to find feasible model assignments.



(a) Remove Assumption A2

(b) JB's model selection

FIGURE E.1: We compare the total serving costs when Assumption A2 is relaxed. We also illustrate the minimal accuracy of the feasible model assignments in JB

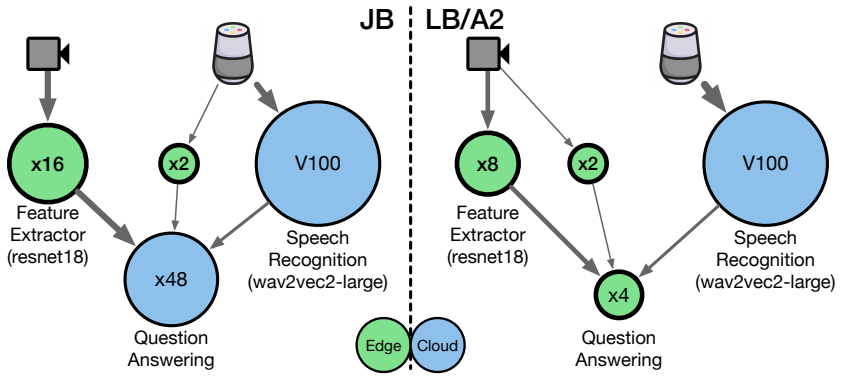
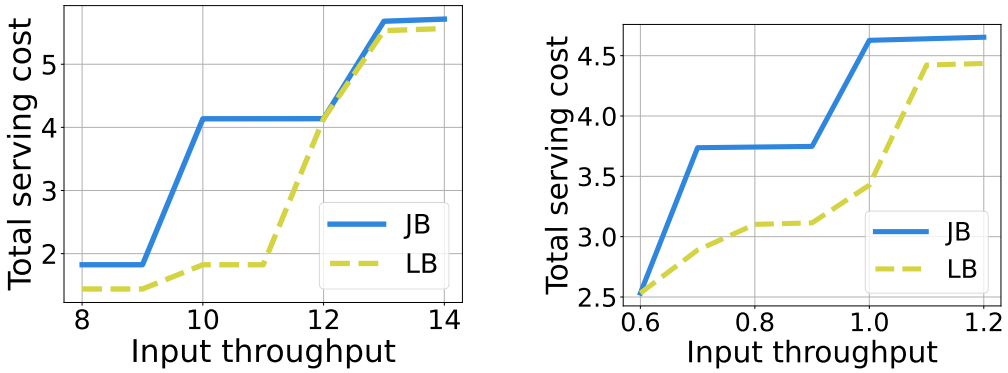


FIGURE E.2: Comparison of the execution plans of JB and LB on VQA using the `medium` setup modified to 25 rps, when Assumption A2 is removed.



(a) VQA (b) AICity

FIGURE E.3: Zoom-in of Figure 5.7 over the regions JB has a noticeably higher cost than LB

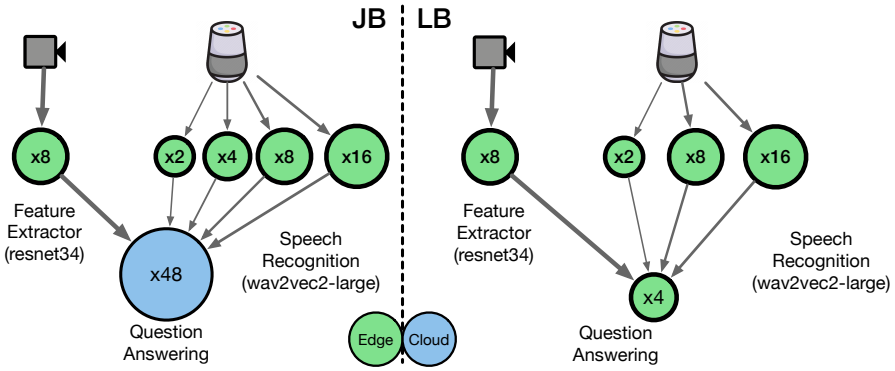


FIGURE E.4: Comparison of the execution plans of JB and LB on VQA using the `medium` setup modified to 10 rps.

E.2 Failure cases in JellyBean

Here we discuss properties of workflow scenarios and infrastructure configurations that can lead to our greedy strategies falling short and thus delivering less-than-optimal execution plans.

First, in some workflows, downstream operators may require less compute than upstream operators (i.e., those closer to the data sources). If the communication cost is not prohibitive, either as a result of the infrastructure cost or the bandwidth requirements, placing the upstream operators on the cloud and downstream operators on the edge may make sense. However, due to our Assumption A2, this placement is prohibited as we assume that information only flows in one direction from lower to higher tiers. Given this assumption, workers on higher tiers should be assigned conservatively as their assignment restricts possible assignments for downstream operators in the workflow. Our greedy strategy shares this philosophy, as it only considers higher tiers when the overall unit cost is cheaper (or when workers on lower tiers are exhausted). In these cases, we find that removing A2 can lead to a reduction in total serving cost for the lower bound. In Figure E.1a, we analyzed the total serving cost of JellyBean (`JB`) compared to a lower bound without A2 (`LB/A2`) using the `medium` setting for VQA. Comparing to Figure 5.7a, we see a larger gap between `JB` and `LB`, as `JB` has up to 66% higher cost than `LB` (among the cases that `LB` outputs different execution plan after removing Assumption A2). We also compare the execution plans generated by `JB` and `LB`, when the assumption is removed. The plans on `medium` setting at 25 rps input throughput are shown in Figure E.2. As we can see, `LB` places the QA operator backward using a x4 worker on edge. Nevertheless, Assumption A2 is still valid in most of the test cases in our main experiments.

Second, in some edge cases, larger models for an operator may not necessarily lead to accuracy improvements. Due to the constant expansion factor in our model selection beam search, `JB` may fail to discover the model with accuracy right above the user-specified threshold. We explore the minimal accuracy of feasible model selections made by `JB` on

AICity in Figure E.1b; we find that in the AICity workflow, the largest ReID model variant's accuracy is consistently lower than some smaller variants.

Third, in some infrastructure setups and workflows, smaller workers may also have lower unit compute cost. JB may then over-provision workers to operators, as it assigns workers by greedily picking the one with the lowest unit cost. In Figure E.3, we zoom in on Figure 5.7 to focus on cases where JB has a higher cost than the lower bound (which are less obvious in the original figure). We found that in these scenarios, JB over-provisions workers to some nodes due to its greedy strategy of iterative worker assignments. A visualization of the assignment in Figure E.4 shows that JB uses one more worker than necessary for speech recognition, since it assigns the x2, x4, and x8 workers before the x16 (thus forcing the question answering operator to execute on the cloud). For CPU workers, this is due to the fact that while the unit cost scales directly with increasing number of cores (i.e., x4 is half the cost of x8), the speedup in execution time does not.

There are some other uncommon scenarios in which JB could generate sub-optimal execution plans. For instance, there might be some workflows where an operator's output accuracy may not monotonically increase with the input accuracy. Under this circumstance, the beam-search strategy in model assignment may fail to discover feasible assignments. Some models may not have stable performance ranking across different workers (e.g., a model executes the fastest compared to other variants on GPU, while being the slowest on CPU); in this case, using a worker-agnostic cost in the model assignment is not sufficient.

Appendix F. Proof of Optimality of the MRO Placement Plan in Lazarus

Recall the setting of our placement problem, we have N nodes, E experts, each node can hold c expert replicas. The $i-$th expert has r_i replicas. Assume there are R nodes alive simultaneously, we want to find a placement plan that maximizes the probability of recovering all the experts when the R alive nodes are sampled uniformly. We denote $[k]$ as the set of $\{1, 2, \cdots, k\}$. We use integer matrix $T \in \mathbb{N}^{c \times N}$ to denote the placement plan, T_{ij} represents the expert placed at node j's i-th slot. T satisfies the following properties:

$$\begin{aligned} &T_{ij} \in [E], \forall i \in [c], j \in [N] \\ &r_k = \sum_{i=1}^{c} \sum_{j=1}^{N} \mathbb{1}_{T_{ij}=k}, \forall k \in [E] \end{aligned} \tag{F.1}$$

Without loss of generality, we assume r is sorted in the ascending order, $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_m$. Let Col_j denote the set composed of elements in the j-th column of T(removing duplicates), $j = 1, \cdots, N$. Let A be the set of R random columns that are alive, A is uniformly sampled. Our goal is:

$$\max \mathsf{Pr}(\bigcup_{a \in A} Col_a = [E]) \tag{F.2}$$

Theorem 1. *The maximum rank overlap placement plan (MRO plan) is defined as follows:* $[N]$ *could be partitioned into* $\lceil \frac{E}{c} \rceil$ *disjoint subsets:* $|S_i| = r_{1+(i-1)*c}, i \in [\lceil \frac{E}{c} \rceil - 1]$, $|S_{\lceil \frac{E}{c} \rceil}| = \min\{N - \sum_{j=1}^{\lceil \frac{E}{c} \rceil - 1} r_{1+(\lceil \frac{E}{c} \rceil - 1)*c}, r_{1+(\lceil \frac{E}{c} \rceil - 1)*c}\}$, *such that, for* $\forall i \in [\lceil \frac{E}{c} \rceil], j \in S_i$, $\{1 + (i-1) * c, \cdots, \min\{i * c, E\}\} \subseteq Col_j$. *We prove that any MRO plan* T *maximizes* $\mathsf{Pr}(\bigcup_{a \in A} Col_a = [E])$.

Proof. We first consider the simple case of $E \leqslant c$.

Under this case, if $N \leqslant r_1 + R - 1$, by Pigeonhole principle, apparently we have $\mathsf{Pr}(\bigcup_{a \in A} Col_a = [E]) = 1$ for any MRO plan.

Otherwise $N \leqslant r_1 + R - 1$, then $|S_1| = r_1$. For any placement plan T, the probability of recovering all experts is upper bounded by the probability of recovering expert 1:

$$\mathsf{Pr}(\bigcup_{a \in A} Col_a = [E]) \leqslant \mathsf{Pr}(1 \in \bigcup_{a \in A} Col_a) \tag{F.3}$$

For any placement plan T, the probability of recovering expert 1 satisfies:

$$\mathsf{Pr}(1 \in \bigcup_{a \in A} Col_a) \leqslant 1 - \frac{\binom{N-r_1}{R}}{\binom{N}{R}} \tag{F.4}$$

For any MRO plan, by definition, we have:

$$\{1, \cdots, E\} \subseteq Col_j, j \in S_1 \tag{F.5}$$

Therefore,

$$\mathsf{Pr}(1 \in \bigcup_{a \in A} Col_a) \geqslant \mathsf{Pr}(\bigcup_{a \in A} Col_a = [E]) \geqslant 1 - \frac{\binom{N-r_1}{R}}{\binom{N}{R}} \tag{F.6}$$

Combining Inequality F.4 and Inequality F.6, we have: for $E \leqslant c$, any MRO plan maximizes $\mathsf{Pr}(\bigcup_{a \in A} Col_a = [E])$ and thus is optimal.

To prove the case of $E > c$, we first define two functions $P_T(\cdot, \cdot, \cdot)$ and $P_s(\cdot, \cdot, \cdot)$. P_T is defined as:

$$P_T(M, n, r) = \mathsf{Pr}(\bigcup_{a \in A} Col_a \supseteq M) \tag{F.7}$$

where matrix $T \in \mathbb{N}^{c \times n}$, A is r columns randomly sampled from n columns , M is a subset of $[E]$. P_T is used to illustrate the probability of recovering the subset M from a sub-matrix T.

For set M, we define $M[j]$ as j-th smallest element in set M. P_s is defined as:

$$P_s(M, n, r) = \mathsf{Pr}(r \text{ samples cover the first } \lceil \frac{|M|}{c} \rceil \text{ segments of vector } v) \tag{F.8}$$

where vector v has length n, with consecutively $\lceil \frac{|M|}{c} \rceil$ segments, the i-th segment has length $L_{M,i} = r_{M[1+(i-1)*c]}, i = 1, \cdots, \lceil \frac{|M|}{c} \rceil - 1, L_{M,\lceil \frac{|M|}{c} \rceil} = \min\{n - \sum_{j=1}^{\lceil \frac{|M|}{c} \rceil - 1} L_{M,j}, r_{\lceil \frac{|M|}{c} \rceil}\}$. P_s is defined to illustrate the recover probability of MRO plans.

We prove the optimality of MRO plan when $E > c$ by mathematical induction. We first have the following assumption:

Assumption 1. $\forall m' < E, \forall n', r', \forall$ *set* $M', |M'| = m'$,

$$\max_T P_T(M', n', r') = P_s(M', n', r') \tag{F.9}$$

We want to prove that for $\forall |M| = E, \forall N, R$,

$$\max_T P_T(M, N, R) = P_s(M, N, R) \tag{F.10}$$

Proving Equation F.10 indicates that any MRO plan achieves optimal recover probability across all different T.

We first consider the case of $|M| > c$. First if $R = 1, |M| > c$, for $\forall T$, $P_T(M, N, R) = 0, P_s(M, N, R) = 0$, the claim trivially satisfies.

When $R > 1, |M| > c$, for $\forall T$, we can transform T to T' by reordering the columns to let the columns containing 1 be the first consecutive columns. And $\forall T$ we have:

$$P_T(M, N, R) = P_{T'}(M, N, R) \tag{F.11}$$

Let A' as the set of R columns randomly sampled on T', S_t be the set of different values of column t of matrix T', C is the largest column ID of T' that contains 1. By conditioning on t, we have:

$$P_{T'}(M, N, R) = \sum_{t=1}^{C} \mathsf{Pr}(\min A' = t) \mathsf{Pr}(\bigcup_{a \in A' \backslash \{t\}} Col_a \supseteq M \backslash S_t | \min A' = t) \tag{F.12}$$

If we consider T'' as the sub-table of T' composed of its last $N - t$ rows, we have:

$$\mathsf{Pr}(\bigcup_{a \in A' \backslash \{t\}} Col_a \supseteq M \backslash S_t | \min A' = t) \leqslant \max_{T''} P_{T''}(M \backslash S_t, N - t, R - 1) \tag{F.13}$$

By Assumption 1, due to $S_t \neq \varnothing$, we have:

$$\max_{T''} P_{T''}(M \backslash S_t, N-t, R-1) = P_s(M \backslash S_t, N-t, R-1) \tag{F.14}$$

Recall Equation F.12, we have:

$$P_{T'}(M, N, R) \leqslant \sum_{t=1}^{r_{M[1]}} \mathsf{Pr}(\min A' = t) P_s(M \backslash S_t, N-t, R-1) \tag{F.15}$$

To upper bound $P_{T'}(M, N, R)$, we have to upper bound $P_s(M \backslash S_t, N-t, R-1)$. We first prove the following proposition:

Proposition 2. *Denote $\mathit{Min}_c M$ as the smallest c elements of M. For $\forall M$, we have:*

$$\mathit{Min}_c M = \arg\max_{S_t} P_s(M \backslash S_t, N-t, R-1) \tag{F.16}$$

It is apparent that removing elements from the recover target set results in an increase of P_s. Therefore, if $|S_t| < c, \forall s \neq S_t$,

$$P_s(M \backslash (S_t \cup s), N-t, R-1) \geqslant P_s(M \backslash S_t, N-t, R-1) \tag{F.17}$$

Therefore the set S_t that maximizes $P_s(M \backslash S_t, N-t, R-1)$ must have c cardinality.

Consider $|S_t| = c$. If S_t is not the smallest c elements of M, we substitute an element in S_t with a smaller element obtaining S'_t, $|S'_t| = c$. By the property of rankings, we have,

$$L_{M \backslash S'_t, i} \geqslant L_{M \backslash S_t, i}, \forall i \tag{F.18}$$

Therefore, $\forall S'_t$ obtained by this way,

$$P_s(M \backslash S'_t, N-t, R-1) \geqslant P_s(M \backslash S_t, N-t, R-1) \tag{F.19}$$

We recursively apply this substitution and obtains $\mathsf{Min}_c M$, therefore, for $\forall S_t$, we have:

$$P_s(M \backslash \mathsf{Min}_c M, N-t, R-1) \geqslant P_s(M \backslash S_t, N-t, R-1) \tag{F.20}$$

Thus finishes the proof of the proposition. This proposition tells us that $S_t = \mathsf{Min}_c M$ maximizes $P_s(M \backslash S_t, N-t, R-1)$.

By Equation F.15 and Proposition 2, we have,

$$P_T(M,N,R) \leqslant \sum_{t=1}^{r_{M[1]}} \mathsf{Pr}(\min A' = t) P_s(M \backslash \mathsf{Min}_c M, N-t, R-1) \tag{F.21}$$

For $P_s(M,N,R)$, consider the left most sample should fall on the first segment, and the other $R-1$ samples should cover the set M', where M' satisfies the j-th segment of M' has equal length with the $j+1$-th segment of M for $\forall j$. Therefore $M' = \{M[1+c], \cdots, M[|M|]\}$.

$$\begin{aligned} P_s(M,N,R) &= \sum_{t=1}^{r_{M[1]}} \mathsf{Pr}(\min A' = t) P_s(M', N-t, R-1) \\ &= \sum_{t=1}^{r_{M[1]}} \mathsf{Pr}(\min A' = t) P_s(\{M[1+c], \cdots, M[|M|]\}, N-t, R-1) \\ &= \sum_{t=1}^{r_{M[1]}} \mathsf{Pr}(\min A' = t) P_s(M \backslash \mathsf{Min}_c M, N-t, R-1) \end{aligned} \tag{F.22}$$

Substituting Equation F.22 into Inequality F.21, we have:

$$P_T(M,N,R) \leqslant P_s(M,N,R) \tag{F.23}$$

Now we have proven that P_s is an upper bound of P_T. Next, we prove that if T is a MRO plan, Inequality F.23 can actually achieve equal. For $\forall$ MRO plan T^*, we have:

$$\bigcup_{a \in A} Col_a = [E] \iff A \text{ covers } S_i, \forall i \in \{1, \cdots, \lceil \frac{E}{c} \rceil\} \tag{F.24}$$

For $\forall$ MRO plan T^*, we can reorder the columns so that for each column set S_i, all columns in S_i are consecutive. We denote the reordered MRO plan as T', and the ran-

domly sampled columns on T' as A'.

$$\begin{aligned}
&\mathsf{Pr}(\bigcup_{a\in A'} Col_a = [E]) \\
=&\mathsf{Pr}(A' \text{ covers segment with length } |S_i|, \forall i \in \{1, \cdots, \lceil \frac{m}{c} \rceil\}) \\
=&P_s(M, N, R)
\end{aligned} \tag{F.25}$$

Therefore for T^* which is a MRO plan, by the definition of P_T in Equation F.7, we have:

$$P_{T*}(M, N, R) = P_s(M, N, R) \tag{F.26}$$

Equation F.26 indicates that $\exists$MRO plan $T^*, P_{T*}(M, N, R) = P_s(M, N, R)$, hence we prove that, under Assumption 1, Equation F.10 holds when $E > c$.

Assumption 1 trivially holds due to the optimality of MRO plan when $E \leqslant c$.

By mathematical reduction, for $\forall E, \forall |M| = E, \forall N, R$, we have,

$$\max_T P_T(M, N, R) = P_s(M, N, R) \tag{F.27}$$

Furthermore, for $\forall$ MRO plan T^* we have:

$$P_{T*}([E], N, R) = \max_T \mathsf{Pr}(\bigcup_{a\in A} Col_a = [E]) \tag{F.28}$$

□

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[2] *AI City Challenge*. `https://www.aicitychallenge.org/2021-ai-city/`. [Last accessed:: 11/17/2022]. 2021.

[3] Mohammad Alizadeh et al. "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters". In: *SIGCOMM*. 2014.

[4] Mohammad Alizadeh et al. "Data Center TCP (DCTCP)". In: *SIGCOMM*. 2010.

[5] Sebastian Angel et al. "End-to-end Performance Isolation Through Virtual Datacenters". In: *OSDI*. 2014.

[6] Stanislaw Antol et al. "Vqa: Visual question answering". In: *IEEE International Conference on Computer Vision (ICCV)*. 2015.

[7] *Apache Arrow*. `https://arrow.apache.org/`. 2022.

[8] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. "Leveraging Service Meshes as a New Network Layer". In: *HotNets*. 2021.

[9] Sanjith Athlur et al. "Varuna: scalable, low-cost training of massive deep learning models". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 472–487.

[10] Jens Axboe. *Efficient IO with io_uring*. `https://kernel.dk/io_uring.pdf`. 2019.

[11] Hitesh Ballani et al. "Enabling End-Host Network Functions". In: *SIGCOMM*. 2015.

[12] Hitesh Ballani et al. "Towards Predictable Datacenter Networks". In: *SIGCOMM*. 2011.

[13] Paul Barham et al. "Pathways: Asynchronous distributed dataflow for ML". In: *arXiv preprint arXiv:2203.12533* (2022).

[14] Andrew Baumann et al. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *SOSP*. 2009.

[15] Adam Belay et al. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". In: *OSDI*. 2014.

[16] Hedi Ben-Younes et al. “Mutan: Multimodal tucker fusion for visual question answering”. In: *IEEE International Conference on Computer Vision (ICCV)*. 2017.

[17] Benjamin Berg et al. “The CacheLib Caching Engine: Design and Experiences at Scale”. In: *OSDI*. 2020.

[18] Brian N. Bershad et al. “Lightweight Remote Procedure Call”. In: *ACM Trans. Comput. Syst.* 8.1 (1990), pp. 37–55. ISSN: 0734-2071. DOI: 10.1145/77648.77650. URL: https://doi.org/10.1145/77648.77650.

[19] Brian N. Bershad et al. “User-Level Interprocess Communication for Shared Memory Multiprocessors”. In: *ACM Trans. Comput. Syst.* (1991).

[20] *Bincode*. https://github.com/bincode-org/bincode. 2022.

[21] Steven Bird. “NLTK: the natural language toolkit”. In: *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*. 2006, pp. 69–72.

[22] Andrew D Birrell and Bruce Jay Nelson. “Implementing remote procedure calls”. In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984), pp. 39–59.

[23] Andrew D. Birrell and Bruce Jay Nelson. “Implementing Remote Procedure Calls”. In: *ACM Trans. Comput. Syst.* (1984).

[24] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. “Accelerating Tensorflow With Adaptive RDMA-Based gRPC”. In: *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 2018.

[25] Daniel Bittman et al. “Don’t Let RPCs Constrain Your API”. In: *HotNets*. 2021.

[26] Tom Brown et al. “Language Models are Few-shot Learners”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.

[27] Zixian Cai et al. “Synthesizing Optimal Collective Algorithms”. In: *PPoPP*. 2021.

[28] Lee Calcote and Zack Butcher. *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O’Reilly Media, 2019.

[29] *Cap’n Proto*. https://capnproto.org/. 2022.

[30] Paris Carbone et al. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[31] Jeffrey S. Chase et al. “Sharing and Protection in a Single-Address-Space Operating System”. In: *TOCS* (1994).

[32] Surajit Chaudhuri. “An overview of query optimization in relational systems”. In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1998, pp. 34–43.

[33] Jingrong Chen et al. “NetHint: White-Box Networking for Multi-Tenant Data Centers”. In: *NSDI*. 2022.

[34] Tianqi Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 578–594.

[35] Youmin Chen, Youyou Lu, and Jiwu Shu. “Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing”. In: *EuroSys*. 2019.

[36] Yu Cheng et al. “A survey of model compression and acceleration for deep neural networks”. In: *arXiv preprint arXiv:1710.09282* (2017).

[37] Zewen Chi et al. “On the representation collapse of sparse mixture of experts”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 34600–34613.

[38] Inho Cho et al. “Overload Control for µs-scale RPCs with Breakwater”. In: *OSDI*. 2020.

[39] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[40] *Consul*. `https://www.consul.io/`. 2022.

[41] Daniel Crankshaw et al. “Clipper: A low-latency online prediction serving system”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[42] Daniel Crankshaw et al. “InferLine: latency-aware provisioning and scaling for prediction serving pipelines”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020.

[43] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. “RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs”. In: *ASPLOS*. 2019.

[44] Michael Dalton et al. “Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization”. In: *NSDI*. 2018.

[45] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. 2004.

[46] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. "Balanced Multicasting: High-Throughput Communication for Grid Applications". In: *ACM/IEEE Conference on Supercomputing (SC)*. 2005.

[47] Jiangfei Duan et al. "Parcae: Proactive,{Liveput-Optimized}{DNN} Training on Preemptible Instances". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 1121–1139.

[48] Abhimanyu Dubey et al. "The llama 3 herd of models". In: *arXiv preprint arXiv:2407.21783* (2024).

[49] *eBPF*. `https://ebpf.io/`. 2022.

[50] *Envoy Proxy*. `https://www.envoyproxy.io/`. 2022.

[51] Steven K Esser et al. "Learned step size quantization". In: *arXiv preprint arXiv:1902.08153* (2019).

[52] *etcd*. `https://etcd.io/`. 2022.

[53] Mohammad Al-Fares et al. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *NSDI*. 2010.

[54] Mohammad Al-Fares et al. "Hedera: dynamic flow scheduling for data center networks." In: *Nsdi*. Vol. 10. 8. San Jose, USA. 2010, pp. 89–92.

[55] William Fedus, Barret Zoph, and Noam Shazeer. "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), pp. 1–39.

[56] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *NSDI*. 2018.

[57] *FlatBuffers*. `https://google.github.io/flatbuffers/`. 2022.

[58] Jason Flinn et al. "Owl: Scale and Flexibility in Distribution of Hot Content". In: *OSDI*. 2022.

[59] Jason Flinn et al. "Owl: Scale and Flexibility in Distribution of Hot Content". In: *OSDI*. 2022.

[60] Rodrigo Fonseca et al. “X-Trace: A Pervasive Network Tracing Framework”. In: *NSDI*. 2007.

[61] Joshua Fried et al. “Caladan: Mitigating Interference at Microsecond Timescales”. In: *OSDI*. 2020.

[62] Yu Gan et al. “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems”. In: *ASPLOS*. 2019.

[63] Wei Gao et al. “Chronus: A novel deadline-aware scheduler for deep learning training jobs”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 609–623.

[64] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[65] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. “In-Network Aggregation for Shared Machine Learning Clusters”. In: *MLSys*. 2021.

[66] *Collective Communications Library with Various Primitives for Multi-Machine Training*. `https://github.com/facebookincubator/gloo`. 2023.

[67] *Gluster*. `https://www.gluster.org/`. 2022.

[68] Joseph E. Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *OSDI*. 2012.

[69] Jianping Gou et al. “Knowledge distillation: A survey”. In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819.

[70] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. “Open MPI: A Flexible High Performance MPI”. In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2005, pp. 228–239.

[71] Robert Grandl et al. “Multi-resource packing for cluster schedulers”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 44.4 (2014), pp. 455–466.

[72] *gRPC*. `https://grpc.io/`. 2022.

[73] *gRPC Release Schedule*. `https://grpc.github.io/grpc/core/md_doc_grpc_release_schedule.html`. 2022.

[74] *gRPC Changelog*. `https://github.com/grpc/grpc/releases`. 2022.

[75] Diandian Gu et al. "ElasticFlow: An elastic serverless training platform for distributed deep learning". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 266–280.

[76] Chuanxiong Guo et al. "RDMA over Commodity Ethernet at Scale". In: *SIGCOMM*. 2016.

[77] Sangtae Ha et al. "TUBE: Time-dependent pricing for mobile data". In: *ACM Special Interest Group on Data Communication (SIGCOMM)*. 2012.

[78] Sangjin Han et al. "MegaPipe: A New Programming Interface for Scalable Network I/O ". In: *OSDI*. 2012.

[79] Mark Handley et al. "Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *SIGCOMM*. 2017.

[80] *HAProxy*. `http://www.haproxy.org/`. 2022.

[81] Olaf Hartmann et al. "Adaptive Selection of Communication Methods to Optimize Collective MPI Operations". In: *PARCO*. 2005.

[82] Jiaao He et al. "Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022, pp. 120–134.

[83] Tao He et al. "Unicron: Economizing self-healing llm training at scale". In: *arXiv preprint arXiv:2401.00134* (2023).

[84] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *NSDI*. 2011.

[85] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. `https://arxiv.org/abs/1704.04861`. 2017. DOI: `10.48550/ARXIV.1704.04861`.

[86] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. "Scrooge: A Cost-Effective Deep Learning Inference System". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 624–638.

[87] Yitao Hu et al. "Rim: Offloading Inference to the Edge". In: *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 2021, pp. 80–92.

[88] Yuzhen Huang et al. "Yugong: Geo-distributed data and job placement at scale". In: *Very Large Data Base Endowment (VLDB)* 12.12 (2019), pp. 2155–2169.

[89] *HuggingFace pre-trained models*. `https://huggingface.co/models`. [Last accessed:: 11/17/2022]. 2022.

[90] Jack Tigar Humphries et al. "GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling". In: *SOSP*. 2021.

[91] Changho Hwang et al. "Elastic resource sharing for distributed deep learning". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 721–739.

[92] Changho Hwang et al. "Tutel: Adaptive mixture-of-experts at scale". In: *Proceedings of Machine Learning and Systems* 5 (2023).

[93] Stephen Ibanez et al. "The nanoPU: A Nanosecond Network Stack for Datacenters". In: *OSDI*. 2021.

[94] *IBM Cloud Pricing*. `https://www.ibm.com/cloud/vpc/pricing`. [Last accessed:: 11/17/2022]. 2022.

[95] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. "PyTorch". In: *Programming with TensorFlow: Solution for Edge Computing Applications* (2021), pp. 87–104.

[96] *Intel MPI Library*. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html`. 2024.

[97] *Istio*. `https://istio.io/`. 2022.

[98] *Rate Limit Policy in Istio*. `https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/`. 2022.

[99] Ethan J. Jackson et al. "SoftFlow: A Middlebox Architecture for Open vSwitch". In: *ATC*. 2016.

[100] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 2704–2713.

[101] Samvit Jain et al. "Scaling video analytics systems to large camera deployments". In: *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. 2019.

[102] Muhammad Asim Jamshed et al. “mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes”. In: *NSDI*. 2017.

[103] Insu Jang et al. “Oobleck: Resilient distributed training of large models using pipeline templates”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 382–395.

[104] Jaeyoung Jang et al. “A Specialized Architecture for Object Serialization with Applications to Big Data Analytics”. In: *ISCA*. 2020.

[105] EunYoung Jeong et al. “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems”. In: *NSDI*. 2014.

[106] Vimalkumar Jeyakumar et al. “EyeQ: Practical Network Performance Isolation at the Edge”. In: *NSDI*. 2013.

[107] Chenyu Jiang et al. “Lancet: Accelerating Mixture-of-Experts Training via Whole Graph Computation-Communication Overlapping”. In: *arXiv preprint arXiv:2404.19429* (2024).

[108] Junchen Jiang et al. “Chameleon: scalable adaptation of video analytics”. In: *ACM Special Interest Group on Data Communication (SIGCOMM)*. 2018.

[109] Yimin Jiang et al. “A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters”. In: *OSDI*. 2020.

[110] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[111] *Introducing JSON*. `https://www.json.org/json-en.html`. 2022.

[112] Anuj Kalia, Michael Kaminsky, and David Andersen. “Datacenter RPCs can be General and Fast”. In: *NSDI*. 2019.

[113] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “Design Guidelines for High Performance RDMA Systems”. In: *USENIX ATC*. 2016.

[114] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs”. In: *OSDI*. 2016.

[115] Svilen Kanev et al. “Profiling a Warehouse-Scale Computer”. In: *ISCA*. 2015.

[116] Daniel Kang, Peter Bailis, and Matei Zaharia. "Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics". In: *arXiv preprint arXiv:1805.01046* (2018).

[117] Yiping Kang et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge". In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 615–629.

[118] Sagar Karandikar et al. "A Hardware Accelerator for Protocol Buffers". In: *MICRO*. 2021.

[119] Nicholas T Karonis et al. "Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance". In: *IPDPS*. 2000.

[120] Georgios P. Katsikas et al. "Metron: NFV Service Chains at the True Speed of the Underlying Hardware". In: *NSDI*. 2018.

[121] Naga Katta et al. "Clove: Congestion-Aware Load Balancing at the Virtual Edge". In: *CoNEXT*. 2017.

[122] Antoine Kaufmann et al. "TAS: TCP Acceleration as an OS Service". In: *EuroSys*. 2019.

[123] Daehyeok Kim et al. "FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds". In: *NSDI*. 2019.

[124] Hyeonji Kim et al. "Natural language to SQL: where are we today?" In: *Very Large Data Base Endowment (VLDB)* 13.10 (2020), pp. 1737–1750.

[125] Joongi Kim et al. "NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors". In: *EuroSys*. 2015.

[126] Marios Kogias et al. "R2P2: Making RPCs first-class datacenter citizens". In: *ATC*. 2019.

[127] Eddie Kohler et al. "The Click Modular Router". In: *ACM Trans. Comput. Syst.* (2000).

[128] Xinhao Kong et al. "Collie: Finding Performance Anomalies in RDMA Subsystems". In: *NSDI*. 2022.

[129] Xinhao Kong et al. "Understanding RDMA Microarchitecture Resources for Performance Isolation". In: *NSDI*. 2023.

[130] *Kubernetes*. `https://kubernetes.io/`. 2022.

[131] Praveen Kumar et al. “PicNIC: Predictable Virtualized NIC”. In: *SIGCOMM*. 2019.

[132] Katrina LaCurts et al. “Choreo: Network-Aware Task Placement for Cloud Applications”. In: *IMC*. 2013.

[133] ChonLam Lao et al. “ATP: In-Network Aggregation for Multi-tenant Learning”. In: *NSDI*. 2021.

[134] Nikita Lazarev et al. “Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs”. In: *ASPLOS*. 2021.

[135] Hung Le et al. “Multimodal transformer networks for end-to-end video-grounded dialogue systems”. In: *arXiv preprint arXiv:1907.01166* (2019).

[136] Jeongkeun Lee et al. “Application-Driven Bandwidth Guarantees in Datacenters”. In: *SIGCOMM*. 2014.

[137] Yunseong Lee et al. “PRETZEL: Opening the black box of machine learning prediction serving systems”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[138] Dmitry Lepikhin et al. “Gshard: Scaling giant models with conditional computation and automatic sharding”. In: *arXiv preprint arXiv:2006.16668* (2020).

[139] Bojie Li et al. “ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware”. In: *SIGCOMM*. 2016.

[140] Jiamin Li et al. “Accelerating distributed {MoE} training and inference with lina”. In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 945–959.

[141] Jiamin Li et al. “Aryl: An elastic cluster scheduler for deep learning”. In: *arXiv preprint arXiv:2202.07896* (2022).

[142] Jing Li et al. “Locmoe: A low-overhead moe for large language model training”. In: *arXiv preprint arXiv:2401.13920* (2024).

[143] Shen Li et al. “Pytorch distributed: Experiences on accelerating data parallel training”. In: *arXiv preprint arXiv:2006.15704* (2020).

[144] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. “HatRPC: Hint-Accelerated Thrift RPC over RDMA”. In: *SC*. 2021.

[145] Yan Li et al. “SpotTune: Leveraging transient resources for cost-efficient hyperparameter tuning in the public cloud”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2020, pp. 45–55.

[146] Yuanqi Li et al. “Reducto: On-camera filtering for resource-efficient real-time video analytics”. In: *ACM Special Interest Group on Data Communication (SIGCOMM)*. 2020.

[147] Zhaogeng Li, Ning Liu, and Jiaoren Wu. “Toward a Production-Ready General-Purpose RDMA-Enabled RPC”. In: *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*. 2019.

[148] Zhuohan Li et al. “Terapipe: Token-level pipeline parallelism for training large-scale language models”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 6543–6552.

[149] Richard Liaw et al. “Hypersched: Dynamic resource reallocation for model development on a deadline”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 61–73.

[150] *Libfabric*. `https://ofiwg.github.io/libfabric/`. 2022.

[151] Tsung-Yi Lin et al. “Microsoft coco: Common objects in context”. In: *European conference on computer vision*. 2014.

[152] *Linkerd*. `https://linkerd.io/`. 2022.

[153] Chong Liu et al. “City-scale multi-camera vehicle tracking guided by crossroad zones”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.

[154] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. “Janus: A unified distributed training framework for sparse mixture-of-experts models”. In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023, pp. 486–498.

[155] Zhuang Liu et al. “Learning efficient convolutional networks through network slimming”. In: *IEEE International Conference on Computer Vision (ICCV)*. 2017.

[156] *Meta Llama 3*. `https://ai.meta.com/blog/meta-llama-3/`. 2024.

[157] Rober Love. *Linux System Programming*. O’Reilly Media, 2007.

[158] Yucheng Low et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. In: *Proc. VLDB Endow.* 5.8 (2012), pp. 716–727. ISSN: 2150-8097. DOI: `10.14778/2212351.2212354`. URL: `https://doi.org/10.14778/2212351.2212354`.

[159] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. “Optasia: A relational platform for efficient large-scale video analytics”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2016.

[160] Yao Lu et al. "Accelerating machine learning inference with probabilistic predicates". In: *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2018.

[161] Liang Luo et al. "PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud". In: *MLSys*. 2020.

[162] Kiwan Maeng et al. "Understanding and improving failure tolerant training for deep learning recommendation with partial recovery". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 637–651.

[163] Christopher D Manning et al. "The Stanford CoreNLP natural language processing toolkit". In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, pp. 55–60.

[164] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. "Cache Craftiness for Fast Multicore Key-Value Storage". In: *EuroSys*. 2012.

[165] Joao Martins et al. "ClickOS and the Art of Network Function Virtualization". In: *NSDI*. 2014.

[166] Michael Marty et al. "Snap: A Microkernel Approach to Host Networking". In: *SOSP*. 2019.

[167] Sarah McClure et al. "Efficient Scheduling Policies for Microsecond-Scale Tasks". In: *NSDI*. 2022.

[168] *Memcached*. `https://memcached.org/`. 2022.

[169] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[170] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. "Edge machine learning for AI-enabled IoT devices: A review". In: *Sensors* 20.9 (2020), p. 2533.

[171] Stephen Merity et al. "Pointer sentinel mixture models". In: *arXiv preprint arXiv:1609.07843* (2016).

[172] Meta. *Meta Llama 3.1 model information*. 2024. URL: `https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md`.

[173] Meta. *Network Observability for AI/HPC Training Workflows*. 2023. URL: `https://atscaleconference.com/videos/network-observability-for-ai-hpc-training-workflows/`.

[174] Meta. *Traffic Engineering for AI Training Networks*. 2023. URL: https://atscaleconference.com/videos/traffic-engineering-for-ai-training-networks/.

[175] Rui Miao et al. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In: *SIGCOMM*. 2017.

[176] Samantha Miller et al. "High Velocity Kernel File Systems with Bento". In: *FAST*. 2021.

[177] Jeffrey C Mogul and John Wilkes. "Nines are not enough: Meaningful metrics for clouds". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 136–141.

[178] Jeffrey C. Mogul and John Wilkes. "Nines Are Not Enough: Meaningful Metrics for Clouds". In: *HotOS*. 2019.

[179] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. "Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock". In: *SOSP*. 2021.

[180] *MongoDB*. https://www.mongodb.com. 2022.

[181] Fabrizio Montesi and Janine Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *arXiv preprint arXiv:1609.05830* (2016).

[182] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *OSDI*. 2018.

[183] Philipp Moritz et al. "Ray: A distributed framework for emerging AI applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[184] *MessagePack*. https://msgpack.org/index.html. 2022.

[185] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[186] Milind Naphade et al. "The nvidia ai city challenge". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE. 2017, pp. 1–6.

[187] Deepak Narayanan et al. "Efficient large-scale language model training on gpu clusters using megatron-lm". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[188] Deepak Narayanan et al. “PipeDream: Generalized pipeline parallelism for DNN training”. In: *Proceedings of the 27th ACM symposium on operating systems principles*. 2019, pp. 1–15.

[189] *The NVIDIA Collective Communication Library (NCCL)*. `https://developer.nvidia.com/nccl`. 2023.

[190] *Nginx*. `https://www.nginx.com/`. 2022.

[191] Xiaonan Nie et al. “Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement”. In: *Proceedings of the ACM on Management of Data* 1.1 (2023), pp. 1–19.

[192] Xiaonan Nie et al. “HetuMoE: An efficient trillion-scale mixture-of-expert distributed training system”. In: *arXiv preprint arXiv:2203.14685* (2022).

[193] NVIDIA. *Performance Reported by NCCL Tests*. `https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md`. 2023.

[194] *NVIDIA TensorRT*. `https://developer.nvidia.com/tensorrt`. [Last accessed:: 11/17/2022]. 2019.

[195] *NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)*. `https://docs.nvidia.com/networking/display/sharpv300`. 2024.

[196] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. “The Akamai Network: A Platform for High-Performance Internet Applications”. In: *SIGOPS Oper. Syst. Rev.* (2010).

[197] Christopher Olston et al. “Tensorflow-serving: Flexible, high-performance ml serving”. In: *arXiv preprint arXiv:1712.06139* (2017).

[198] Vladimir Olteanu et al. “Stateless Datacenter Load-balancing with Beamer”. In: *NSDI*. 2018.

[199] *oneAPI Collective Communications Library (oneCCL)*. `https://github.com/oneapi-src/oneCCL`. 2023.

[200] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *ATC*. 2014.

[201] Amy Ousterhout et al. “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads”. In: *NSDI*. 2019.

[202] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Vol. 2. Springer, 1999.

[203] Siddharth Pal et al. "Efficient All-to-All Collective Communication Schedules for Direct-Connect Topologies". In: *arXiv preprint arXiv:2309.13541* (2023).

[204] Shoumik Palkar et al. "E2: A Framework for NFV Applications". In: *SOSP*. 2015.

[205] Aurojit Panda et al. "NetBricks: Taking the V out of NFV". In: *OSDI*. 2016.

[206] Rina Panigrahy et al. *Heuristics for vector bin packing*. `https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/`. 2011.

[207] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).

[208] Y Peng et al. "A Generic Communication Scheduler for Distributed DNN Training Acceleration". In: *SOSP*. 2019.

[209] Simon Peter et al. "Arrakis: The Operating System is the Control Plane". In: *OSDI*. 2014.

[210] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *NSDI*. 2015.

[211] Jelena Pješivac-Grbović et al. "MPI Collective Algorithm Selection and Quadtree Encoding". In: *Parallel Computing* (2007).

[212] Maksym Planeta et al. "MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications". In: *ATC*. 2021.

[213] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *arXiv preprint arXiv:1802.05668* (2018).

[214] Salvatore Pontarelli et al. "FlowBlaze: Stateful Packet Processing in Hardware". In: *NSDI*. 2019.

[215] Lucian Popa, Ali Ghodsi, and Ion Stoica. "HTTP as the Narrow Waist of the Future Internet". In: Hotnets-IX. 2010.

[216] Lucian Popa et al. "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing". In: *SIGCOMM*. 2013.

[217] Arash Pourhabibi et al. "Cerebros: Evading the RPC Tax in Datacenters". In: *MICRO*. 2021.

[218] Arash Pourhabibi et al. “Optimus Prime: Accelerating Data Transformation in Servers”. In: *ASPLOS*. 2020.

[219] Leon Poutievski et al. “Jupiter Evolving: Transforming Google’s Datacenter Network via Optical Circuit Switches and Software-Defined Networking”. In: *SIGCOMM*. 2022.

[220] George Prekas, Marios Kogias, and Edouard Bugnion. “ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks”. In: *SOSP*. 2017.

[221] *Protocol Buffers*. `https://developers.google.com/protocol-buffers`. 2022.

[222] Qifan Pu et al. “Low latency geo-distributed data analytics”. In: *ACM SIGCOMM Computer Communication Review (CCR)* 45.4 (2015), pp. 421–434.

[223] *PyTorch pre-trained models*. `https://pytorch.org/vision/stable/models.html`. [Last accessed:: 11/17/2022]. 2022.

[224] Aurick Qiao et al. “Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning”. In: *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 2021.

[225] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[226] Sivasankar Radhakrishnan et al. “SENIC: Scalable NIC for End-Host Rate Limiting”. In: *NSDI*.

[227] Deepti Raghavan et al. “Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather”. In: *HotOS*. 2021.

[228] Deepti Raghavan et al. “Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather”. In: *HotOS*. HotOS ’21. 2021.

[229] Costin Raiciu et al. “Improving Datacenter Performance and Robustness with Multipath TCP”. In: *SIGCOMM*. 2011.

[230] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. “CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters”. In: *NSDI*. 2024.

[231] Samyam Rajbhandari et al. “Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale”. In: *International conference on machine learning*. PMLR. 2022, pp. 18332–18346.

[232] Saeed Rashidi et al. “Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models”. In: *ISCA*. 2022.

[233] Jeff Rasley et al. “Deepspeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 3505–3506.

[234] *ROCm Communication Collectives Library (RCCL)*. `https://github.com/ROCm/rccl`. 2023.

[235] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[236] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)* 28 (2015).

[237] Francisco Romero et al. “INFaaS: A model-less inference serving system”. In: *arXiv preprint arXiv:1905.13348* (2019).

[238] Francisco Romero et al. “Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2021.

[239] Benjamin Rothenberger et al. “ReDMArk: Bypassing RDMA Security Mechanisms”. In: *USENIX Security*. 2021.

[240] Hugo Sadok et al. “We Need Kernel Interposition over the Network Dataplane”. In: *HotOS*. 2021.

[241] *Use Managed Spot Training in Amazon SageMaker*. `https://docs.aws.amazon.com/sagemaker/latest/dg/model-managed-spot-training.html`. 2024.

[242] Russel Sandberg. “The Sun Network File System: Design, Implementation and Experience”. In: *USENIX Summer ATC*. 1986.

[243] Victor Sanh, Thomas Wolf, and Alexander Rush. “Movement pruning: Adaptive sparsity by fine-tuning”. In: *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), pp. 20378–20389.

[244] Amedeo Sapio et al. “Scaling Distributed Machine Learning with In-Network Aggregation”. In: *NSDI*. 2021.

[245] Mike Schroeder and Michael Burrows. "Performance of Firefly RPC". In: *ACM Transaction on Computer Systems* (1990). URL: https://www.microsoft.com/en-us/research/publication/performance-of-firefly-rpc/.

[246] SemiAnalysis. *100,000 H100 Clusters: Power, Network Topology, Ethernet vs InfiniBand, Reliability, Failures, Checkpointing*. 2024. URL: https://www.semianalysis.com/p/100000-h100-clusters-power-network.

[247] SemiAnalysis. *The Inference Cost Of Search Disruption – Large Language Model Cost Analysis*. 2023. URL: https://www.semianalysis.com/p/the-inference-cost-of-search-disruption.

[248] Daniele De Sensi et al. "Swing: Short-cutting Rings for Higher Bandwidth Allreduce". In: *NSDI*. 2024.

[249] Aashaka Shah et al. "TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches". In: *NSDI*. 2023.

[250] Haichen Shen et al. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2019.

[251] Shaohuai Shi et al. "ScheMoE: An Extensible Mixture-of-Experts Distributed Training System with Tasks Scheduling". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 236–249.

[252] Mohammad Shoeybi et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[253] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010.

[254] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[255] Arjun Singhvi et al. "1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters". In: *SIGCOMM*. 2020.

[256] Arjun Singhvi et al. "CliqueMap: Productionizing an RMA-Based Distributed Caching System". In: *SIGCOMM*. 2021.

[257] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable Cross-Language Services Implementation". In: *Facebook white paper* 5.8 (2007), p. 127.

[258] Shaden Smith et al. “Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model”. In: *arXiv preprint arXiv:2201.11990* (2022).

[259] *Snabb: Simple and fast packet networking*. `https://github.com/snabbco/snabb`. 2022.

[260] Marco Spuri and Giorgio C. Buttazzo. “Efficient aperiodic service under earliest deadline scheduling”. In: *Real-Time Systems Symposium*. 1994, pp. 2–11.

[261] Patrick Stuedi et al. “DaRPC: Data Center RPC”. In: *SoCC*. 2014.

[262] Maomeng Su et al. “RFP: When RPC is Faster than Server-Bypass with RDMA”. In: *EuroSys*. 2017.

[263] Mark Sutherland et al. “The NEBULA RPC-Optimized Architecture”. In: *ISCA*. 2020.

[264] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *CVPR*. 2016.

[265] Mingxing Tan and Quoc Le. “Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks”. In: *ICML*. 2019.

[266] Puqi Perry Tang and Tsung-Yuan Charles Tai. “Network Traffic Characterization Using Token Bucket Model”. In: *INFOCOM*. 1999.

[267] Zheng Tang et al. “Cityflow: A city-scale benchmark for multi-target multi-camera vehicle tracking and re-identification”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[268] John Thorpe et al. “Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}”. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 497–513.

[269] *Apache Thrift*. `https://thrift.apache.org/`. 2022.

[270] *Timely Dataflow*. `https://github.com/TimelyDataflow/timely-dataflow`. [Last accessed:: 11/17/2022]. 2015.

[271] *Timely dataflow map function*. `https://docs.rs/timely/latest/timely/dataflow/operators/map/trait.Map.html`. 2015.

[272] *Tonic*. `https://github.com/hyperium/tonic`. 2022.

[273] *TorchElastic*. https://pytorch.org/docs/stable/distributed.elastic.html. 2024.

[274] Ankit Toshniwal et al. "Storm@twitter". In: *SIGMOD*. 2014.

[275] *Triton Inference Server*. https://github.com/triton-inference-server/server. [Last accessed:: 11/17/2022]. 2021.

[276] Shin-Yeh Tsai and Yiying Zhang. "LITE Kernel RDMA Support for Datacenter Applications". In: *SOSP*. 2017.

[277] Uber. *DataMesh: How Uber laid the foundations for the data lake cloud migration*. 2024. URL: https://www.uber.com/blog/datamesh/.

[278] Uber. *Up: Portable Microservices Ready for the Cloud*. 2023. URL: https://www.uber.com/blog/up-portable-microservices-ready-for-the-cloud.

[279] Amin Vadhat. *Coming of Age in the Fifth Epoch of Distributed Computing: The Power of Sustained Exponential Growth*. Amin Vahdat - SIGCOMM Lifetime Achievement Award 2020 Keynote. 2020. URL: %5Curl%7Bhttps://www.youtube.com/watch?v=27zuReojDVw%7D.

[280] *Virtual GPU (vGPU) NVIDIA*. https://www.nvidia.com/en-us/data-center/virtual-solutions/. [Last accessed:: 11/17/2022]. 2022.

[281] Can Wang et al. "Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics". In: *IEEE Conference on Computer Communications (INFOCOM)*. 2020.

[282] Guanhua Wang et al. "Blink: Fast and Generic Collectives for Distributed ML". In: *MLSys*. 2020.

[283] Shibo Wang et al. "Overlap communication with dependent computation via decomposition in large deep learning models". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2022, pp. 93–106.

[284] Stephanie Wang et al. "Ownership: A Distributed Futures System for Fine-Grained Tasks". In: *NSDI*. 2021.

[285] Yuxin Wang et al. "Reliable and Efficient In-Memory Fault Tolerance of Large Language Model Pretraining". In: *arXiv preprint arXiv:2310.12670* (2023).

[286] Zhuang Wang et al. "Gemini: Fast failure recovery in distributed training with in-memory checkpoints". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 364–381.

[287] Matt Welsh and David Culler. "Adaptive Overload Control for Busy Internet Servers". In: *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*. 2003.

[288] *What's Inside Our New DNNCam? Learn About The Hardware.* `https://boulderai.com/whats-inside-our-new-dnncam-learn-about-the-hardware/`. [Last accessed:: 11/17/2022]. 2022.

[289] Wired. *OpenAI's CEO Says the Age of Giant AI Models Is Already Over*. 2023. URL: `https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/`.

[290] Thomas Wolf et al. "Transformers: State-of-the-art natural language processing". In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.

[291] Adam Wolnikowski et al. "Zerializer: Towards Zero-Copy Serialization". In: *HotOS*. 2021.

[292] Wencong Xiao et al. "{AntMan}: Dynamic scaling on {GPU} clusters for deep learning". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 533–548.

[293] Jiali Xing et al. "Charon: A Framework for Microservice Overload Control". In: *HotNets*. 2021.

[294] Jiarong Xing et al. "Bedrock: Programmable Network Support for Secure RDMA Systems". In: *USENIX Security*. 2022.

[295] Zhihui Yang et al. "Optimizing Machine Learning Inference Queries with Correlative Proxy Models". In: *arXiv preprint arXiv:2201.00309* (2022).

[296] Zongheng Yang et al. "{SkyPilot}: An intercloud broker for sky computing". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 437–455.

[297] *YOLOv5 releases*. `https://github.com/ultralytics/yolov5/releases/tag/v6.0`. [Last accessed:: 11/17/2022]. 2021.

[298] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *NSDI*. 2012.

[299] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[300] Mingshu Zhai et al. “{SmartMoE}: Efficiently Training {Sparsely-Activated} Models through Combining Offline and Online Parallelization”. In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 961–975.

[301] Chengliang Zhang et al. “Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving”. In: *USENIX Annual Technical Conference (ATC)*. 2019.

[302] Hong Zhang et al. “Resilient Datacenter Load Balancing in the Wild”. In: *SIGCOMM*. 2017.

[303] Irene Zhang et al. “The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems”. In: *SOSP*. 2021.

[304] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. “Gallium: Automated Software Middlebox Offloading to Programmable Switches”. In: *SIGCOMM*. 2020.

[305] Susan Zhang et al. “Opt: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022).

[306] Wuyang Zhang et al. “Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading”. In: *ACM Conference on Mobile Computing and Networking (MobiCom)*. 2021.

[307] Yazhuo Zhang et al. “LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing”. In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 2023, pp. 502–519.

[308] Yiwen Zhang et al. “Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks”. In: *NSDI*. 2022.

[309] Liangyu Zhao et al. “Optimal Direct-Connect Topologies for Collective Communications”. In: *arXiv preprint arXiv:2202.03356* (2022).

[310] Lianmin Zheng et al. “Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 559–578.

[311] Pengfei Zheng et al. “Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning”. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 703–723.

[312] Hao Zhou et al. “Overload Control for Scaling WeChat Microservices”. In: *SoCC*. 2018.

[313] Yanqi Zhou et al. “Mixture-of-experts with expert choice routing”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 7103–7114.

[314] Xiangfeng Zhu et al. “Dissecting Service Mesh Overheads”. In: *ArXiv* abs/2207.00592 (2022).

[315] Yibo Zhu et al. “Congestion Control for Large-Scale RDMA Deployments”. In: *SIGCOMM*. 2015.

[316] Danyang Zhuo et al. “Slim: OS Kernel Support for a Low-Overhead Container Overlay Network”. In: *NSDI*. 2019.

[317] Barret Zoph et al. “St-moe: Designing stable and transferable sparse expert models”. In: *arXiv preprint arXiv:2202.08906* (2022).

[318] Simiao Zuo et al. “Taming sparsely activated transformer with stochastic experts”. In: *arXiv preprint arXiv:2110.04260* (2021).