University of Texas at Arlington

MavMatrix

Computer Science and Engineering Dissertations

Computer Science and Engineering Department

2023

RESOURCE PROVISIONING FOR DATA-INTENSIVE USER-FACING APPLICATIONS

Huiyang Li

Follow this and additional works at: https://mavmatrix.uta.edu/cse_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Li, Huiyang, "RESOURCE PROVISIONING FOR DATA-INTENSIVE USER-FACING APPLICATIONS" (2023). *Computer Science and Engineering Dissertations*. 327.
https://mavmatrix.uta.edu/cse_dissertations/327

This Dissertation is brought to you for free and open access by the Computer Science and Engineering Department at MavMatrix. It has been accepted for inclusion in Computer Science and Engineering Dissertations by an authorized administrator of MavMatrix. For more information, please contact leah.mccurdy@uta.edu, erica.rousseau@uta.edu, vanessa.garrett@uta.edu.

RESOURCE PROVISIONING FOR DATA-INTENSIVE USER-FACING APPLICATIONS

by

HUIYANG LI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
Aug 2023

RESOURCE PROVISIONING FOR DATA-INTENSIVE USER-FACING APPLICATIONS

The members of the Committee approve the doctoral dissertation of Huiyang Li

Hao Che
Supervising Professor ____________________

Hong Jiang
Co-supervising Professor ____________________

Yonghe Liu ____________________

Jia Rao ____________________

Dean of the Graduate School ____________________

Copyright © by Huiyang Li 2023
All Rights Reserved

To my family,
for their constant support and unconditional love.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude and appreciation to all those who have contributed to the completion of this thesis during my Ph.D. study at UT Arlington. This research would not have been possible without their unwavering support and encouragement.

First and foremost, I am deeply thankful to my supervisor, Dr. Hao Che, and my co-supervisor Dr. Hong Jiang, for their invaluable guidance, mentorship, and constant encouragement throughout this research endeavor. Their expertise, dedication, and constructive feedback have played a pivotal role in shaping this work, and I am immensely grateful for their continued support.

I extend my heartfelt thanks to the members of my thesis committee, Dr. Yonghe Liu and Dr. Jia Rao, for their time, expertise, and thoughtful evaluation of this work. Their constructive criticisms and valuable suggestions have undoubtedly elevated the quality of this thesis.

My lab research professor Dr. Zhijun Wang and my lab mates, Dr. Nguyen Minh, Dr. Zhongwei Li, Dr, Ning Li, Lin Sun, Todd Rosenkrantz, Prathyusha Enganti, Xuan Wang, and Akshit Singhal deserve a special mention for their camaraderie and the stimulating academic environment they helped create. Their willingness to engage in insightful discussions and offer assistance has been invaluable during this research journey.

To my dear friends who meet at UT Arlington, Chaochao Yan, Chen Zhong, Lingfeng Xiang, Jian Li, and Xingsheng Zhao, I am deeply appreciative of your unwavering belief in me and your constant encouragement. Your motivation and emotional support were a driving force that kept me going even during challenging times.

I cannot forget to acknowledge the unwavering support of my mother and other family members. Their love, understanding, and encouragement have been a source of strength throughout my academic pursuits. I am forever indebted to them for their sacrifices and belief in my abilities.

Last but not least, I would like to extend my gratitude to all my friends and well-wishers who have supported me in various ways during this journey.

In conclusion, this thesis is a culmination of the collective efforts and encouragement of the aforementioned individuals. Their contributions have had a profound impact on the success of this work, and for that, I am truly thankful.

Jul 20, 2023

ABSTRACT

RESOURCE PROVISIONING FOR DATA-INTENSIVE USER-FACING APPLICATIONS

Huiyang Li, Ph.D.
The University of Texas at Arlington, 2023

Supervising Professor: Hao Che & Hong Jiang

Data-intensive, User-facing Services (DUSes) such as web searching, digital marketing, online social networking, and online retailing are critical workloads in clouds and datacenters. Meeting stringent query tail-latency Service Level Objectives (SLO) for DUS queries is essential for optimal user experience and business success. However, achieving these objectives is challenging due to the scale-out nature of DUese workloads and the varying resource demands of queries with different fanouts. Additionally, the design and configuration options for clusters significantly impact query performance.

In this dissertation, we present solutions of DUSes performance online and offline optimization. We highlight the importance of reducing query tail latency and the impact on user experience and revenue. We discuss the complexities of meeting tail-latency SLOs considering query fanout and the need to allocate resources accordingly. Furthermore, we explore the wide range of cluster design and configuration options and propose model-based approaches to compare and identify promising configurations.

Through queuing models, we establish the maximum sustainable cluster loads and analyze worker and cluster-level performance. We validate our models through extensive simulation and testing, providing valuable insights for DUSes design and efficient resource planning. Our work contributes to improving user experience, resource optimization and resource provisioning plan in cloud-based DUSes environments.

Overall, our online solution optimized/guaranteed the tail latency while improve resource utilization, and our offline models analysis and findings provide guidance for DUSes service providers, enabling enhanced user experience and effective resource provisioning.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v
ABSTRACT vi

Chapter Page

1. INTRODUCTION 1
   1.1 Optimize the user-facing service's latency and maintain high resource utilization 1
   1.2 Guarantee the tail latency SLO for DUSes 2
   1.3 Address different cluster design and configuration options for DUSes 3
   1.4 Initial resource provisioning tool 4
   1.5 Organization 5
2. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler 6
   2.1 Introduction 6
   2.2 Pigeon Scheduler 9
      2.2.1 System model 9
      2.2.2 Task Scheduling 10
   2.3 Performance Modeling and Analysis 12
   2.4 Simulation Testing 17
      2.4.1 Parameter Settings 17
      2.4.2 Performance evaluation 22
   2.5 Performance Evaluation on EC2 Cloud 26
   2.6 Practical Considerations 27
      2.6.1 Master Failure Recovery 28
      2.6.2 Dealing with Heterogeneous Workers and Tasks with Assignment Constraints 28
   2.7 Related Work 29
   2.8 Conclusions 31
   2.9 Acknowledgments 31
3. TailGuard: Tail Latency SLO Guaranteed Task Scheduling for Data-Intensive User-Facing Applications 32
   3.1 Introduction 32
   3.2 Background and Related Work 35
      3.2.1 Data-Intensive User-Facing Services 35
      3.2.2 Tail Latency Aware Solutions for DU Services 37

3.3 TailGuard . . . 38
3.3.1 TailGuard Query Processing Model . . . 39
3.3.2 Task Queuing Deadline Estimation . . . 41
3.3.3 Query admission control . . . 45
3.4 Performance Evaluation . . . 45
3.4.1 Workloads . . . 46
3.4.2 Impact of query fanout . . . 47
3.4.3 Impact of service class . . . 50
3.4.4 Joint with Outlier Alleviation Solution . . . 52
3.4.5 TailGuard with Query Admission Control . . . 53
3.4.6 Evaluation in the Amazon EC2 cloud . . . 54
3.4.7 Evaluation in an SaS Testbed . . . 56
3.5 Conclusions . . . 58
4. Performance Models for Data-intensive, User-facing Workloads with Query Tail Latency SLO . . . 60
4.1 Introduction . . . 60
4.2 A Unified Model . . . 62
4.2.1 Model for Cluster Scaling . . . 63
4.2.2 Models for Resource Scaling . . . 67
4.2.3 Performance Bounds . . . 71
4.2.4 Task Response time distribution: $F_T(t)$ . . . 73
4.3 Numerical Analysis . . . 75
4.3.1 Model accuracy evaluation . . . 75
4.3.2 Crossover analysis . . . 80
4.4 Related Work . . . 80
4.5 Conclusions . . . 83
5. Initial Resource Provisioning Plan Tool . . . 85
5.1 Introduction . . . 85
5.2 Model . . . 86
5.3 Methodology . . . 87
5.3.1 System description . . . 87
5.3.2 Input Variables and Requirements . . . 88
5.4 Case Study . . . 90
5.4.1 Predefined requirements and variables . . . 90
5.4.2 Worker level input variables . . . 90
5.4.3 Cluster level input variables . . . 91
5.5 Conclusion . . . 92

6. CONCLUSIONS AND FUTURE WORK 93
6.1 Contributions 93
6.2 Future Work 94
REFERENCES 95
BIOGRAPHICAL STATEMENT 104

CHAPTER 1
INTRODUCTION

Data-intensive, User-facing Services (DUSes) such as web searching, digital marketing, online social networking, and online retailing have become predominant workloads in clouds and datacenters. Meeting stringent query tail-latency Service Level Objectives (SLO) for DUS queries is crucial for ensuring optimal user experience, application adoption, customer satisfaction, and business revenue. However, achieving these objectives presents challenges due to the large scale of DUSes workloads, where queries may need to access massive data sets distributed across a large number of servers. In other words, a query may spawn a number of tasks (known as query fanout) to be sent to some or all of the workers in the cluster to be queued and processed in parallel. The query response time is determined by the slowest task of the query. Furthermore, the resource demands for queries with different fanouts vary, making it essential to consider query expansion when allocating resources. Additionally, there is a wide range of possible cluster design and configuration options and different options may lead to vastly different query tail latency and throughput performance. Besides the above feature, in today's datacenters, the workload is heterogeneous which means the DUSes are served with long batch jobs to help improve datacenter resource utilization. Therefore, it's a long-standing challenge in datacenter scheduling.

This dissertation aims to investigate both online techniques for optimizing and guaranteeing query tail latency and improving throughput and resource utilization, as well as offline performance models and tools for providing initial resource provisioning plans.

1.1 Optimize the user-facing service's latency and maintain high resource utilization

Workload heterogeneity poses challenges for schedulers to simultaneously meet latency requirements and maintain high resource utilization. Workloads that differ in execution time and fanout degree have distinct requirements for scheduling. While short jobs are usually user-facing application [19, 63] have stringent latency requirements and are sensitive to scheduling delays; long jobs such as batch jobs help improve datacenter resource utilization and can tolerate some scheduling delays. The state-of-the-art schedulers, including centralized, distributed, and hybrid ones, struggle to ensure low latency for short

jobs in large-scale, highly loaded systems. The main issues lie in the scalability of centralized schedulers and the ineffective probing and resource sharing in distributed and hybrid schedulers. Addressing these challenges and achieving low latency for short jobs while optimizing resource utilization are key research objectives in datacenter scheduling.

We propose Pigeon, a distributed, hierarchical job scheduler based on a two-layer design that can significantly optimize the tail latency of the user-facing services online. Pigeon divides workers into groups, each managed by a separate master. In Pigeon, upon a job arrival, a distributed scheduler directly distributes tasks evenly among masters with minimum job processing overhead, hence, preserving the highest possible scalability. Meanwhile, each master manages and distributes all the received tasks centrally, oblivious of the job context, allowing for full sharing of the worker pool at the group level to maximize multiplexing gain. To minimize the chance of head-of-line blocking for short jobs and avoid starvation for long jobs, two weighted fair queues are employed in each master to accommodate tasks from short and long jobs, separately, and a small portion of the workers are reserved for short jobs. Evaluation via theoretical analysis, trace-driven simulations, and a prototype implementation shows that Pigeon significantly outperforms Sparrow up to 150X, a representative distributed scheduler, and Eagle up to 30X, a hybrid scheduler at 90th percentile tail latency for short jobs(user-facing services).

1.2 Guarantee the tail latency SLO for DUSes

It has been widely recognized that the query tail latency for Data-intensive User-facing (DU) services, such as web searching, online social networking, and emergency response through edge-based crowdsensing, has a great impact on user experience and hence, business revenues. For example, for Amazon online web services, every 100-millisecond addition of query tail latency causes 1% decrease in sales [**?**]. To meet strict tail latency Service Level Objectives (SLOs), the resources for DU services are generally over-provisioned [26, 12], at the cost of reduced profit. As a result, a key design objective of a DU service, called **the design objective** in short hereafter, *is to maximize the resource utilization or query throughput while meeting tail latency SLOs for individual queries*. Unfortunately, the existing solutions fall short of achieving this design objective, which we argue, is largely attributed to the fact that they fail to take the query fanout explicitly into account. For example, assume that with a given amount of resources allocated to process each task and the task response time for each task has 1% probability to be over 100 ms. Then the query response time for a query with fanout k_f has a probability, 1-0.99^{k_f}, to be over 100 ms, meaning that a query with k_f=1 and k_f=100 have 1% and 63.4% probabilities of being over 100 ms, respectively. This implies that while a query with k_f=1 can meet the

tail latency SLO in terms of the 99th percentile tail latency of 100 ms, a query with k_f=100 cannot. In order to allow the query with k_f=100 to also meet the same tail latency SLO, a task associated with the query must be allocated a much larger amount of resource so that the chance it will exceed 100 ms is as small as 0.01%. This ensures that the probability that the query response time exceeds 100 ms is 1-0.9999^{100} = 0.01 or 1%, i.e., meeting the same tail latency SLO as the query with k_f=1. This example clearly demonstrates that to meet a query tail latency SLO for all queries regardless of query fanouts, the task resource demands for tasks belonging to queries with different fanouts are different and a task belonging to a query with a larger fanout demands more resources

To address the challenge of the design objective of a DUSes, we propose TailGuard, a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing(TF-EDFQ) policy, as a first step towards achieving the design objective for DU services in general. In other words, on top of optimizing the latency, we further guarantee the tail latency. As a top-down approach, TailGuard decouples the upper query level design from the lower task level design. First, at the query level, a task decomposition technique is developed to translate the query tail latency SLO for a query with a given fanout into a task queuing deadline for tasks spawned by the query at the task level, reflecting the resource demand of the tasks. This effectively decomposes a hard cotask scheduling problem at the query level into individual queue management subproblems at the task level. Second, at the task level, a single TF-EDFQ corresponding to a task server is used to enforce the task queuing deadlines, as a way to differentiate resource allocation for tasks with different resource demands. In principle, TailGuard permits unlimited number of query classes and is lightweight, as it incurs minimum overhead for task queuing deadline estimation and requires to implement only a single earliest-deadline-first queue per task server for any DU applications. A query admission control scheme is also developed to provide a tail latency SLO guarantee in the face of resource shortages.

The experiment results demonstrate that TailGuard can improve resource utilization by up to 80%, while meeting the targeted tail latency SLOs, as compared with the other three policies: First-In-First-Out (FIFO) task queuing, task PRIority Queuing (PRIQ), and Tail-latency-SLO-aware EDFQ (T-EDFQ) policies.

1.3 Address different cluster design and configuration options for DUSes

The first challenge to achieve the above design objective is failing to take the query fanout explicitly into account as mentioned in the above section. Second, there are a wide range of possible cluster design and configuration options and different options may lead to vastly different query tail latency and throughput performance. We identify some widely

adopted design and configuration choices, in terms of, scale-up or scale-out, with or without redundant task issues [126] at the worker level; and vertical scaling or horizontal scaling with or without tail cutting [61] at the cluster level. This leads to a total of 16 distinct design and configuration options. It becomes apparent that it is impractical to design and configure a cluster to exhaust all the options and then identify which one should be adopted to achieve the above design objective. Instead, one must resort to model-based approaches that can help quickly compare different possible options to identify the most promising few before field trial.

We develop queuing models that provide a direct connection between cluster resource demand, query tail-latency SLO and throughput for all design and configuration options. It derives the maximum sustainable cluster loads at different query tail-latency-to-mean ratios for different design and configuration options; under certain resource scaling conditions, there is a worker-level cross-over load, independent of query fanout and there is a cluster-level cross-over load, a function of query fanout. By combining these performance models it can assist decision-makers in reducing the number of options to consider when designing and configuring clusters, thereby improving efficiency and achieve the design objective goal.

The accuracy of the proposed models in predicting the DUS performance is verified by simulation that details can be found in Section 4.3.

1.4 Initial resource provisioning tool

With the cloud computing evolution, traditional on-premise computing is increasingly being left behind. A large number of organizations are trying to migrate their existing workflows and applications to the cloud. The average cloud server pricing is about $400 monthly for one server, to $15,000 monthly for the entire back-office infrastructure. This is the cost of a rental server which doesn't include other costs such as maintenance, and operation. We call the purchase price plus the costs of operation TCO. In conclusion, it is very important to have an offline initial resource planning tool to help users who need to deploy their services into the cloud with minor changes after the services are fully deployed and running and meet the user tail latency SLO requirements in the meantime.

We developed an initial resource provisioning tool based on our performance models. This tool is measurement based, in that user provides some initial measurement information of their DUSes input into the tool, as the output it will give the user an overview of the tail latency at different loads at both worker and cluster levels. Users can use this output as a reference to choose their initial plan with their special requirements. We use our highly heterogeneous Sensing-as-a-Service (SaS) testbed to do an experiment, explain

how the tool with proposed models make it possible to decouple the worker-level decision making from the cluster-level decision making, hence, substantially reducing the number of possible design and configuration options to be compared as a use case of our tool.

1.5 Organization

The rest of the dissertation is organized as follows. In Chapter 2, we introduce the Pigeon scheduler, an effective distributed, hierarchical datacenter job scheduler that optimized the user-facing application tail latency online. In Chapter 3, we present TailGuard: tail latency SLO guaranteed task scheduling for Data-Intensive User-Facing applications. In Chapter 4, we propose offline performance models for Data-intensive, User-facing workloads with query tail latency SLO improving initial resource provisioning plan efficiency and achieving the design objective goal. In Chapter 5, We develop an initial resource provisioning tool with the performance model to help users build their DUSes in the cloud. In Chapter 6, we summarize our contributions and discuss future work.

CHAPTER 2

Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler

In today's datacenters, job heterogeneity makes it difficult for schedulers to simultaneously meet latency requirements and maintain high resource utilization. The state-of-the-art datacenter schedulers, including centralized, distributed, and hybrid schedulers, fail to ensure low latency for short jobs in large-scale and highly loaded systems. The key issues are the scalability in centralized schedulers, ineffective and inefficient probing and resource sharing in both distributed and hybrid schedulers.

In this paper, we propose Pigeon, a distributed, hierarchical job scheduler based on a two-layer design. Pigeon divides workers into groups, each managed by a separate master. In Pigeon, upon a job arrival, a distributed scheduler directly distribute tasks evenly among masters with minimum job processing overhead, hence, preserving highest possible scalability. Meanwhile, each master manages and distributes all the received tasks centrally, oblivious of the job context, allowing for full sharing of the worker pool at the group level to maximize multiplexing gain. To minimize the chance of head-of-line blocking for short jobs and avoid starvation for long jobs, two weighted fair queues are employed in each master to accommodate tasks from short and long jobs, separately, and a small portion of the workers are reserved for short jobs. Evaluation via theoretical analysis, trace-driven simulations, and a prototype implementation shows that Pigeon significantly outperforms Sparrow, a representative distributed scheduler, and Eagle, a hybrid scheduler.

2.1 Introduction

Workload heterogeneity has been a long-standing challenge in datacenter scheduling. Jobs that differ in execution time and fanout degree have distinct requirements for scheduling. Short jobs have stringent latency requirements and are sensitive to scheduling delays; long jobs, which usually have a large fanout and high resource demands, require high-quality scheduling, e.g., improving load balance, but can tolerate some scheduling delays. While short jobs are usually user-facing applications [19, 63] and important to user-perceived quality-of-service, long jobs help improve datacenter resource utilization.

Therefore, it is common practice to collocate short and long jobs in datacenter management, but meeting the diverse needs of heterogeneous jobs remains a critical challenge.

Early datacenter job schedulers, e.g., Jockey [44], Quincy [60], Tetrished [123], Delay Scheduling [139], Firmament [46] and Yarn [125] are centralized by design. Centralized schedulers rely on a global view of resource availability to make scheduling decisions. As systems scale, handling a large number of jobs and collecting runtime status from a large number of nodes inevitably become a bottleneck and incur a significant scheduling delay for each job. This is particularly problematic for short jobs with tight deadlines.

To address the scalability issue, recent research, such as Sparrow [91] and Peacock [73], employs multiple schedulers to dispatch tasks in an independent and distributed manner. Without requiring a global view of resources, distributed schedulers probe randomly selected nodes (usually twice as many as the number of tasks to be dispatched) and dispatch tasks onto the least loaded nodes. The probe based technique has been proved to greatly improve task queuing time compared to random placement [91]. However, each scheduler still needs to maintain a fairly large amount of probe related states and incurs non-negligible probe processing overheads.

Besides the above issues, the collocation of heterogeneous workloads presents unique challenges to the centralized and distributed schedulers. First, heterogeneous workloads require an effective mechanism to prioritize short jobs over long jobs. Distributed schedulers lack coordination among one another, thereby unable to enforce global service differentiation among jobs. While centralized schedulers can employ priority queues to differentiate task scheduling for different types of jobs, they are usually work conserving – low priority, long jobs can utilize the entire cluster to avoid wasting cluster resources. However, by doing so, a burst of long jobs can inflict the so called head-of-line blocking to short jobs that arrive immediately after the burst. Even in the presence of centralized priority queues, tasks from short jobs need to wait for the tasks of long jobs that have already been dispatched onto workers. Recent work BigC [22] and Karios [35] propose to suspend long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks. Second, high resource utilization in datacenters that embrace workload consolidation makes randomized load balancing less effective. For heterogeneous workloads that contain tasks of various sizes, it is difficult to identify less loaded nodes. It has also been reported that randomized load balancing is inefficient and requires multiple rounds of probing to locate idle or less busy nodes if most nodes are highly loaded [115].

Hybrid approaches, such as Mercury [69], Hawk [36] and Eagle [34], combine centralized and distributed schedulers, with former handling long jobs and the latter short jobs. However, long and short jobs are scheduled independently. This makes it difficult to miti-

gate the negative impact of long jobs on the performance of short jobs. For example, Eagle [34] employs two techniques to entirely eliminate the head-of-line blocking, i.e., multiple rounds of probing for short-job task placement and a reserved worker pool for short jobs. However, as the cluster load becomes high, most of the short jobs are driven by long jobs to the reserved pool [35], resulting in rapid performance deterioration for short jobs. Our simulations based on the Yahoo trace [25] show that the performance of short jobs drastically degrades, by as many as 70 times at high load compared with the non-resource-constrained case (see Section 4 for details).

In this paper, we demonstrate that a hierarchical scheduler that employs a divide-and-conquer approach in task scheduling can effectively overcome the shortcomings of centralized, distributed and hybrid schedulers, and ensure low latency for short jobs while maintaining high resource utilization without significantly sacrificing the performance of long jobs. Specially, we propose *Pigeon*, a two-layer, hierarchical scheduler for heterogeneous jobs. Pigeon divides workers in a cluster into groups and delegates task scheduling in each group to a group master. Upon job submission, Pigeon assigns the tasks of an incoming job to the masters as evenly as possible. The dispatching of tasks onto masters is intended to be simple and does not consider the type of tasks. The master in each group implements more sophisticated scheduling by maintaining two weighted fair queues, one for tasks from short jobs and the other for tasks from long jobs, respectively, and partitioning workers in each group into high and low priority workers. Tasks of short jobs can run on any workers while tasks of long jobs can only run on low priority workers. Tasks are only dispatched when there are idle workers from a group and are otherwise queued at a respective priority queue according to their types.

Pigeon is a hierarchical solution purposely designed for effective task distribution to combat heterogeneity. Pigeon's two-layer design is specially useful for heterogeneous jobs. First, it effectively mitigates head-of-line blocking of short jobs. The simple job-oblivious task dispatching among masters prevents a burst of tasks from monopolizing all workers and provides a certain level of isolation between jobs. Unlike in a centralized scheduler, where tasks of the same type (e.g., short jobs) are usually served in FIFO order, tasks of different jobs in Pigeon are evenly distributed among masters, allowing tasks that arrive late to start to execute even before some tasks of an earlier job start to execute (see Section 4.1 for details). Second, the two-layer design preserves good scalability of distributed schedulers but avoids the pitfalls of randomized load balancing. The size- and type-oblivious task dispatching among masters provide sufficient randomness for effective load balancing without global knowledge and the weighted fair queuing based scheduling within a group is deterministic, ensuring that idle workers are rapidly located to serve latency-sensitive jobs without starving the long jobs.

We perform an evaluation of Pigeon through theoretical analysis, simulation, and a prototype implementation on the Amazon EC2 cloud. Analysis results show that Pigeon can greatly increase the job-zero-queuing probability compared to Sparrow, a representative distributed scheduler, for workloads that only contain short jobs. Trace-driven simulations based on the Yahoo, Cloudera and Google traces demonstrate that Pigeon outperforms Eagle, a state-of-the-art hybrid scheduler, on short job performance by as many as tens of times in a highly loaded cluster. Experimental results on the Amazon EC2 further confirm the effectiveness of Pigeon.

2.2 Pigeon Scheduler

This section presents Pigeon. We first give an overview of Pigeon and introduce its task placement scheme, and then discuss how it handles tasks at the master level.

2.2.1 System model

We consider a datacenter cluster composed of a large number of workers, each of which can be an independent processing unit, such as a CPU core. The workers can run in parallel to execute different tasks. A key idea in Pigeon is to divide workers into groups. Each group is managed by a master which centrally controls all the tasks handled by the group and places tasks among the workers in the group. Distributed job schedulers directly distribute the tasks belonging to a job to the masters. After a master receives a task, it either directly sends the task to an idle worker to be processed immediately or puts it in the corresponding task queue if there is no idle worker in the group at the time. Figure 2.1 gives a system overview of Pigeon. The system is composed of multiple distributed job schedulers, masters, and workers. All job schedulers work independently and do not exchange any task placement information among themselves.

A master works at the task level and is mostly job oblivious except for its awareness of whether a task is a low or high priority one, based on whether the task is from a short or a long job. It maintains two weighted fair task queues, where the high priority and low priority queues store tasks belonging to short and long jobs, respectively. The classification of a job as a short or long job is handled by schedulers, based on the type of application the job belongs to. For example, user-facing applications, such as web searching and social networking, that generally have short task execution times and require stringent tail latency guarantee, can be classified as short jobs. On the other hand, background batch applications, such as data backup, that usually have long task execution times and call for loose mean response time assurance, can be classified as long jobs. In Pigeon, a small

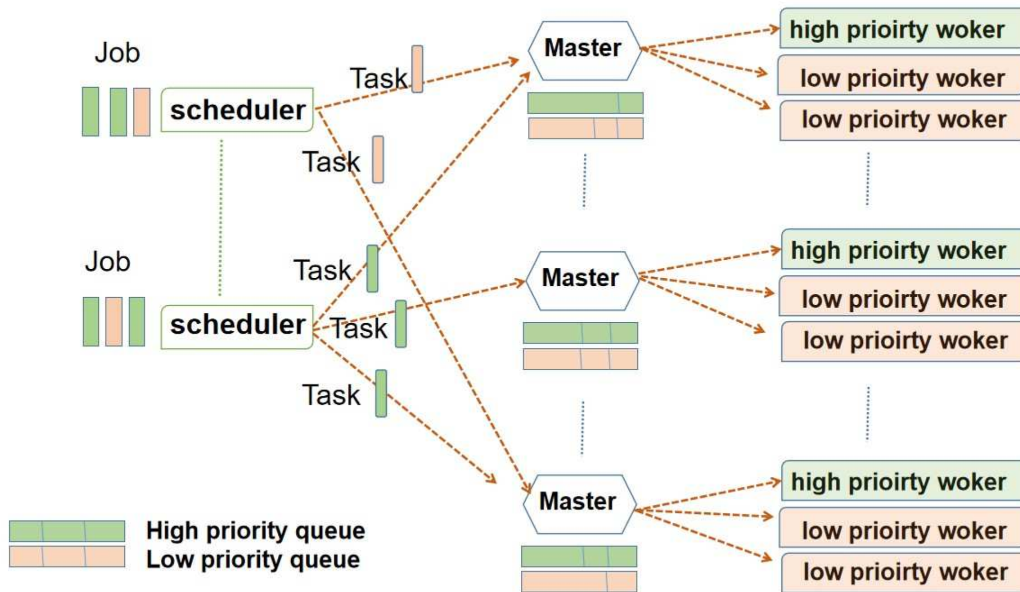


Figure 2.1: Overview of Pigeon.

number of workers in the group, called high priority workers, are reserved exclusively for serving high priority tasks. The other workers, called low priority workers, can serve both low and high priority tasks. Since all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs, Pigeon can greatly improve resource efficiency, achieving high multiplexing gain, compared with the existing job schedulers that distribute tasks directly to individual workers.

A master can run in a worker who needs to be be allocated enough computation resource to effectively handle group status report and task placement functions. As we shall discuss in more detail later, in Pigeon a master needs to handle about one incoming task per second on average, which is modest from computation resource demand point of view.

2.2.2 Task Scheduling

Assume that a system has N_s schedulers and N_g groups (i.e., N_g masters). Each group has N_w workers in it. For a job with F tasks (i.e., fanout degree, F), the scheduler that handles the job will distribute the tasks as follows. It sends $S = \lfloor F/N_g \rfloor$ task(s) to each master (here $\lfloor x \rfloor$ represents the floor of x, i.e., the integer part of x) and the remaining $r = F\%N_g$ to r randomly selected masters. Since the number of workers in each group is

much larger than one (i.e., in the range of 50 to 100), according to the law of large numbers, the workloads distributed to different groups are expected to be much more balanced than those distributed directly to individual workers. This helps synchronize the task processing for tasks belonging to the same job and hence, reduce the job completion time, with respect to the existing job scheduling solutions.

Two task queues of different scheduling priorities are set in each master to store the corresponding types of tasks[1], i.e., tasks belonging to short and long jobs. More specifically, the two queues are scheduled based on weighted fair queuing with a single integer weight to ensure that tasks from the high priority queue are served with higher priority than those from the low priority queue, without starving the low priority tasks. The queue scheduler ensures that out of every W tasks to be served, at least one comes from the low priority task queue if it is not empty. The queue scheduler degenerates to strict priority queuing, when W is set to infinity. In this case, the low priority tasks can be served only when the high priority task queue is empty.

A master maintains two idle worker lists, i.e., the high and low priority idle worker lists that record all high and low priority workers that are currently idle, respectively. A task sent to a master must include the priority of the task. When a master receives a high priority task, it first checks whether the low priority idle worker list is empty or not. If the list is not empty, an idle worker from the list is removed and assigned to handle the task. Otherwise, the master checks whether the high priority idle worker list is empty or not. If it is not empty, a worker is removed from the list and assigned to handle the task. If both idle worker lists are empty, the high priority task is put into the high priority task queue. When a master gets a low priority task, it only checks the low priority idle worker list. If the list is not empty, a worker is removed from the list to serve the task. Otherwise, the task is put into the low priority task queue. Whenever a worker is selected to handle a task, the master sends the task to the worker, together with the scheduler identifier (ID) for the scheduler from which the task is received. If a master receives multiple tasks from a job at a time, it handles these tasks one by one consecutively following the same procedure.

We note that both reserving a given portion of workers in a group for high priority tasks and setting W to be a finite integer help to avoid head-of-line-blocking of short jobs and starvation of long jobs, respectively. The exact values of these two parameters must be properly selected in practice. For all our real-world-trace-driven case studies (see Section 4), we found that no more than 10% of workers need to be reserved to achieve high short job performance, lower than that of Eagle, a state-of-the-art hybrid scheduler. In the meantime, W can be simply set to infinity to achieve the highest short job performance without

[1]Pigeon can be easily extended to support more than two job types by allocating as many priority queues as the number of job types with weighted fair queuing.

significantly impacting the long job performance. This is because the trace statistics show that the short job execution time is less than 20% of the overall job execution time and hence, long jobs have little chance to be starved by short jobs.

When a worker completes a task, it sends the reports/results directly to the corresponding scheduler and meanwhile, sends an idle notification message to its master. This may further trigger a task in one of the two queues to be sent to the worker or the worker to be added to the high priority worker list if it is reserved for high priority jobs, otherwise, to the low priority worker list.

2.3 Performance Modeling and Analysis

To gain insights on the Pigeon performance, in this section, we conduct simple performance modeling and analyses for Pigeon, compared with the analysis of a performance model for Sparrow [91]. To be mathematically tractable, we consider only one class of jobs and assume that the job fanout degree (i.e., the number of tasks in a job) is less than the number of groups and the number of workers in Pigeon and Sparrow, respectively. Hence, only one task queue is used in each master. In this case, all the workers serve tasks from all jobs. We focus on short jobs, which are usually more latency sensitive and whose fanout degrees are smaller than long jobs. We assume that the job fanout degrees are no larger than the number of groups.

Consider a cluster with N_g groups and each group with N_w workers, with a total of $N_c = N_g N_w$ workers in the cluster. Assume that jobs arrive following a Poisson arrival process with average arrival rate λ. All the jobs have fanout degree, F, where $F \leq N_g$, and the task execution time follows an exponential distribution with average execution time, T_e.

With the above model, each master can then be approximately modeled as running a single M/M/N_w task queue [29] with average task arrival rate $\lambda_t = \lambda F / N_g$. The worker utilization is $\rho = \lambda_t T_e / N_w$. Given that $F \leq N_g$, the probability, $P_{task}(0)$, that a task experiences zero queuing time in a group is then given as follows [29],

$$P_{task}(0) = 1 - \frac{1}{1 + (1-\rho)(\frac{N_w!}{(N_w \rho)^{N_w}}) \sum_{k=0}^{N_w - 1} \frac{(N_w \rho)^k}{k!}}, \tag{2.1}$$

and the average queuing time T_q for a task in a master is

$$T_q = \frac{1 - P_{task}(0)}{N_w / T_e - \lambda_t}. \tag{2.2}$$

In this paper, a job is considered to have zero queuing time if the job completion time (not including the communication time) is equal to its longest task execution time. For example, assume that a job has 2 tasks with execution time 10s and 100s, respectively. If the job completion time is 100s, it experiences no queuing delay, even though its task with 10s execution time may have queued for some time, e.g., 50s.

Now we first consider the case that all the tasks in a job have the same execution time. Then the job-zero-queuing probability in Pigeon, $P_{job}^{PI}(0)$, can be written as,

$$P_{job}^{PI}(0) = (P_{task}(0))^F. \tag{2.3}$$

In this case, the job-zero-queuing probability for Sparrow, $P_{job}^{SP}(0)$, using $2F$ probes per job, is derived in the original paper on Sparrow [91], as follows,

$$P_{job}^{SP}(0) = \sum_{i=F}^{2F}(1-\rho)^i \rho^{2F-i} C(2F, i), \tag{2.4}$$

where $C(2F, i)$ is the combination function.

Figure 2.2 depicts the analytical job-zero-queuing probability for Sparrow (i.e., Eqn.(2.4)) and Pigeon (i.e., Eqn.(2.3)) for two different group sizes, i.e., N_w=100 and 200 and two job fanout degrees, i.e., $F = 50$ and 100. As one can see, the job-zero-queuing probability for Sparrow starts to drop at load 0.4 and quickly drops to near zero at load 0.6, whereas for Pigeon, similar drops occur in a much higher load region, i.e., 0.6 to 0.8. It means that Pigeon can work at 20 - 40% higher load than Sparrow, while achieving similar job-zero-queuing performance as Sparrow, demonstrating the effectiveness of Pigeon for job scheduling, compared with Sparrow.

We also note that for Pigeon, when N_w increases from 100 to 200, the job-zero-queuing probability starts to drop at load 0.7, 0.1 higher than the former case. But it quickly approaches 0 as the load approaches 0.9, similar to the former case. This suggests that a larger group can improve performance in the medium load region (0.7 to 0.8), but not much in high load region (>0.9).

The above analyses assume that each task in a job has the same execution time. However, real trace analyses indicate that the task execution time can vary significantly from one task to another for a given job. To capture the performance impact of such variability, we consider the case where the task execution time for a task in a given job follows an exponential distribution.

We first calculate the average job queuing time, T_{job}. Since the job queuing time is defined as the queuing time of the slowest task of the job, we need to find the queuing time

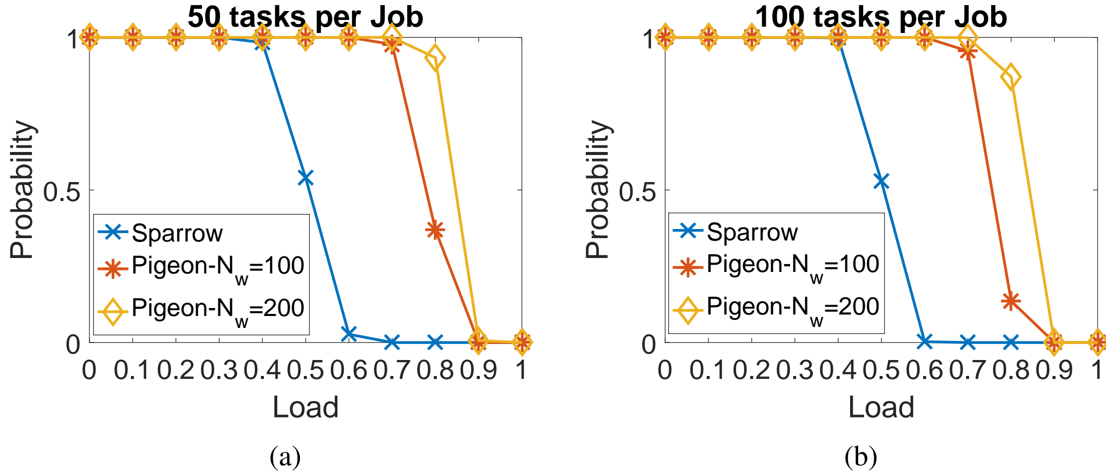


Figure 2.2: Job-zero-queuing probabilities for Sparrow and Pigeon with two different group sizes (N_w=100 and 200). All tasks in a job have the same execution time. (a) Job fanout F=50; (b) F=100.

for the slowest task of the job. To this end, we observe from figs/pigeon 2.3(b) and 3.6(b) that the average queuing time in Pigeon is much shorter than the average task execution time (i.e., T_e=100 ms) even at a high load (e.g., 90%). This suggests that whichever task has the largest execution time is likely to be the slowest one, regardless of its queuing time. This implies that the average queuing time for the slowest task can be simply approximated as the average queuing time for all tasks, i.e., $T_{job} \approx T_q$.

Now we calculate the job-zero-queuing probability. Consider two independent exponential distribution random variables (t_1 and t_2) with average value T_e, the joint probability density function $f(t_1, t_2) = \frac{1}{T_e^2} e^{-(t_1+t_2)/T_e}$. Then the probability of $t_1 - t_2 > T_q$ under condition $t_1 > t_2$ [106] is

$$P(t_1 - t_2 > T_q | t_1 > t_2) = e^{-T_q/T_e}. \tag{2.5}$$

Let $A1$ and $A2$ be the tasks with the longest (t_1) and second longest (t_2) execution times in a job, respectively. Now the job-zero-queuing probability $P_{job}^{dt}(0)$ for a job with different task execution times can be approximately expressed as the probability of $A1$ with zero-queuing time (i.e., $P_{task}(0)$) while $t_1 - t_2 > T_q$, i.e., the execution time difference between the longest and the second longest task execution time is greater than the average task queuing time T_q, namely,

$$P_{job}^{dt}(0) \approx P_{task}(0) e^{-T_q/T_e}. \tag{2.6}$$

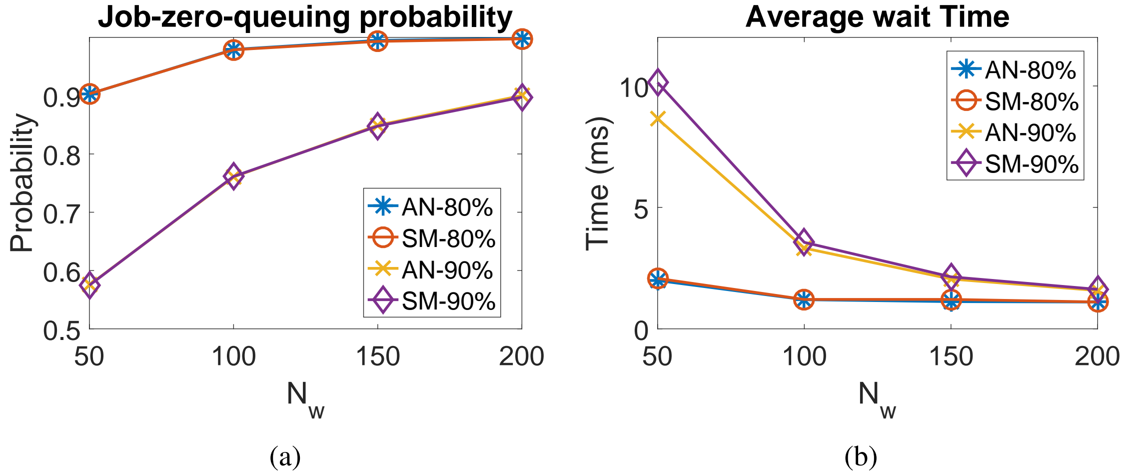


Figure 2.3: Analysis (denoted as AN) vs Simulation (denoted as SM) at different group sizes. (a) Job-zero-queuing Probabilities (b) Average job wait time.

We verify the analytical approximations for T_{job} and P_{job}^{dt} by simulation. Assume that N_c=30,000, $F = 100$, and T_e=100 ms. Each task execution time follows an exponential distribution. The communication time is set at 0.5 ms between any two nodes. We note that with communication delay, the average job waiting time T_w is no longer equal to the average queuing time, but rather the average queuing time plus the communication time.

We study the Pigeon performance by changing N_w from 50 to 200 (the total number of workers N_c in the system is fixed). Figure 2.3 depicts the job-zero-queuing probability and the average job waiting time at two different high loads (i.e., 80% and 90%). We note that the simulation results (denoted as SM) closely match the analytical ones (denoted as AN), e.g., less than 1% for the job-zero-queuing probability for all N_w's tested. The largest difference is about 12% for the average waiting time at N_w=50 and the load of 90%. In this case, the simulated waiting time (also queuing time) is longer than the analytical one because the analytical results only consider the waiting time for the task with the longest execution time. As the job-zero-queuing probability is low (below 60%), the contribution of other tasks may not be neglected, resulting in larger errors.

The results verify that Eqns. (2.2) and (2.6) can be used to estimate the performance of Pigeon for handling jobs with fanout degrees less than the number of groups. The results indicate that the job-zero-queuing probability increases and the average waiting time decreases as the group size increases. It means that a larger group can provide better performance, particularly from 50 to 100. The performance improves slower as the group size increases from 100 to 200, particularly for the average waiting time. Further increasing the group size is expected to offer marginal performance gain. This result provides some

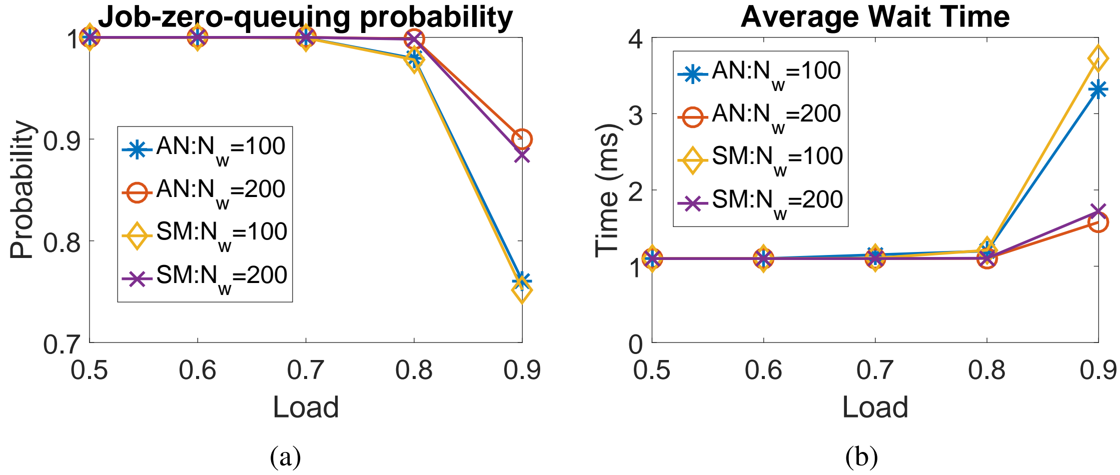


Figure 2.4: Analysis (denoted as AN) vs Simulation (denoted as SM) results with various cluster loads. (a) Job-zero-queuing probabilities (b) Average job wait time.

Trace	F_{max}	F_{min}	F_{avg}	T_e^{max} (s)	T_e^{min} (s)	T_e^{avg} (s)
Yahoo	5900	1	39.91	21259.9	1.54E-5	118.78
Cloudera	51834	1	272.93	97941.8	3.89E-5	162.19
Google	49960	1	35.32	774922	1E-6	661.74

Table 2.1: Trace statistics of job fanout and execution time

insight on how to set the right group size when a cluster handles jobs with small fanout degree (i.e., the number of tasks in a job is less than the number of groups in the cluster).

Now we study the performance of Pigeon by varying cluster loads. Two cases with N_w set at 100 and 200 are studied. The results are given in Figure 3.6. Again, the simulation results closely match the analytical ones. The results indicate that the job-zero-queuing probability is close to 1 even at load 80% and reduces to 0.7 at load 90%. This means that most jobs do not need to be queued even at high load, hence offering high probability of meeting the tightest job performance requirements at high load. We also note that the average waiting time is very small (less than 4%) compared to the average task execution time even at very high load, e.g., 90%. These results clearly demonstrate the effectiveness of Pigeon for job scheduling.

The following two sections test the efficiency of Pigeon by large-scale simulation and on a small EC2 cloud cluster, respectively.

2.4 Simulation Testing

To test the scalability and efficiency, we use simulation to evaluate the performance of Pigeon against Eagle[2] in large clusters, using three real-world traces as input, i.e., Yahoo [25], Cloudera [24], and Google traces [98]. The open source simulation code of Eagle [34] is used and an event-driven simulator is developed for Pigeon.

Table 2.1 provides the statistics of these traces, including the maximum/minimum/average job fanout degrees (denoted as $F_{max}/F_{min}/$ F_{avg}) and maximum/minimum/average task execution time (denoted as T_e^{max}/T_e^{min} $/T_e^{avg}$). We see that the job fanout degree ranges from 1 to 51834; the execution time varies from microseconds to over 700K seconds; and the average task execution time ranges from 118.78 seconds to 661.74 seconds. Unlike the modeled workload in the previous section, these statistics indicate that the job size in terms of both fanout degree and task execution time vary significantly from job to job in practice. Such job heterogeneity makes it difficult to meet service requirements for individual applications, e.g., in terms of providing job completion time or throughput guarantee. For example, for a cluster with 10K workers and a long job with fanout degree of 50K, each worker needs to execute 5 tasks for the job on average. The placement of such a job evenly among all the workers in the cluster can take up all the cluster resources at once, causing head-of-line blocking to the upcoming short jobs. As aforementioned, to effectively deal with the job heterogeneity issue, both Pigeon and Eagle [34] reserve a subset of workers to be used by short jobs only, at the group-level and cluster-level, respectively. In what follows, we first discuss the parameter settings, in terms of the short-vs-long job thresholds, the reserved worker pool size, the communication delays, the group size, and the weight value for weighted fair queuing and then performance evaluation.

2.4.1 Parameter Settings

Short Jobs vs Long Jobs: As mentioned earlier, in practice, a scheduler may rely on whether a job belongs to a user-facing application to classify it as a short job or not. However, due to the lack of the application information for the three traces and to fairly compare against Eagle, for Pigeon, we simply use the same short job cutoff times, defined as the average task execution time of a job, as those used in Eagle, i.e., 90.5811, 272.783 and 1129.532 seconds for the Yahoo, Cloudera and Google traces, respectively.

Reserved Worker Pool Size: The actual number of workers reserved for tasks of short jobs has significant impact on job completion times for both short and long jobs. The more workers are reserved, the smaller the job completion time for short jobs but the larger the

[2] As Eagle outperforms Sparrow and Hawk[36], only Eagle is compared here.

job completion time for long jobs. We study the performance using the three traces by varying the worker reservation ratio (due to page limitation, the results are not presented here). By taking into account of the performance for both short and long jobs, we decide to set the reservation ratios at 2%, 7% and 9% for the Yahoo, Cloudera and Google traces, respectively. For Eagle, against which Pigeon is to be compared in the following section, we set the reservation ratios for the three traces at the same values as those used in [34], i.e., 2%, 9% and 17%, respectively.

The reservation ratio that gives the best performance tradeoffs for Pigeon is between 2%-9% for the three traces. We also found that setting this ratio at 5% for all the traces leads to a maximal performance deviation from the best tradeoffs within 20% for both short and long jobs. Hence, to address the possible lack of the traces in practice, the ratio can be initially set at 5% and then adjusted as the trace workload runs long enough to estimate the best ratio.

Weight Value for Fair Queuing: The weight value W is an important parameter for Pigeon. A smaller (larger) W helps improve the performance of long (short) jobs at the cost of the other. We study the Pigeon performance by varying W from 5 to 100 and compared to that with strict priority queuing (i.e. W is set to infinity) (again, the results are not presented here due to page limitation). We find that the short job performance becomes very sensitive to W at high cluster loads when W gets below 20. For example, while the 99th-percentile short job completion time at $W = 20$ is within 140% of that at $W = \infty$, it increases to more than 300% at $W = 5$, at high cluster loads for all the three traces. Meanwhile, we find that the long job performance is insensitive to W in a wide range, e.g., only 2% difference from $W = 10$ to ∞ at all cluster loads for all the three traces. In other words, no long job starvation occurs even at $W = \infty$ for all the three traces. Hence, for all the cases studied, we set W in the range of 20 to ∞.

Communication Delays: The communication delays are set at 0.5 ms between any two nodes, i.e., a scheduler and a master, a master and a worker, or a worker and a scheduler.

Group Size: Without knowing the exact processing overhead per task scheduling at each master, we have not taken this overhead into account in both performance modeling in the previous section and the simulation in this section. As a result, intuitively, one would expect that the testing results in both previous and this section will be always in favor of larger group size, with the group size equal to the cluster size offering the highest performance (i.e., the case when Pigeon degenerates to a centralized scheduler). While this intuition is confirmed in the previous section based on the results from an ideal model, much to our

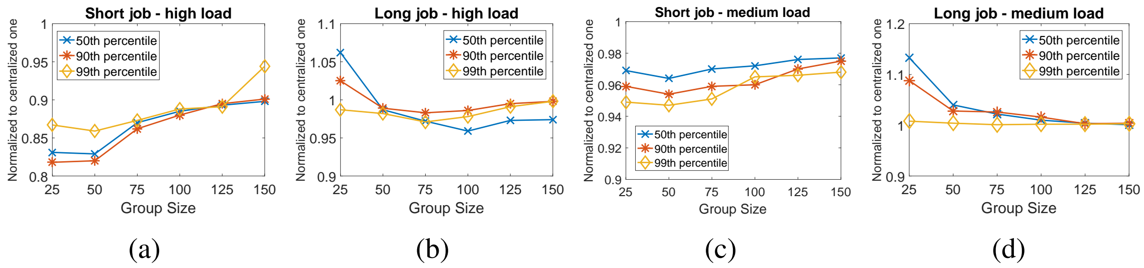


Figure 2.5: Pigeon performance at different group sizes (Pigeon is normalized to the centralized scheduler).

surprise, it turns out to be false as confirmed by the simulation results in this section. More specifically, we can conclude that *Pigeon with the group size in a finite range actually outperforms its centralized counterpart, even when the centralized scheduler incurs negligible processing overhead.* The implication of this is significant. It means that one can no longer assume that so long as it is free from scalability concerns, centralized scheduling is always the best choice, as it has a global view of the cluster resource availability. In what follows, we first identify the range and the preferred group size, and then provide an explanation of why this seemingly counter-intuitive phenomenon can happen.

We compare the Pigeon performance at different group sizes, using the Cloudera trace as input for the simulation (similar results are obtained for the Yahoo and Google traces and hence are not given here). We consider the cluster size of 12K and 18K, corresponding to high (about 93%) and medium (about 62%) cluster loads, respectively. All other parameters pertaining to Pigeon are the same as those given above. The 50th, 90th and 99th percentiles of the short and long job completion times are used as performance metrics.

From the results depicted in Figure 2.5 (normalized to the centralized one), we can see that Pigeon performs better than its centralized counterpart for all the three performance metrics for short jobs, particularly at the high load (figs/pigeon 2.5 (a)). At high load, the short job performance gets better as the group size reduces from 150 to 50 and then becomes slightly worse as it further reduces to 25. The largest performance gains for short jobs are about 17%, 18% and 14% for 50th, 90th and 99th percentile job completion times at group size 50, compared to the centralized one, respectively. Similar results with smaller gains are observed at the medium cluster load (Figure 2.5 (c)). For long jobs at high load (Figure 2.5 (b)), the performance is better (worse) than the centralized one when group size is above (below) 50. The three percentiles of job completion times decrease when the group size reduces from 150 to 100, and then increase when the group size further reduces. In the medium load (Figure 2.5 (d)), the performance for long jobs is worse than that of

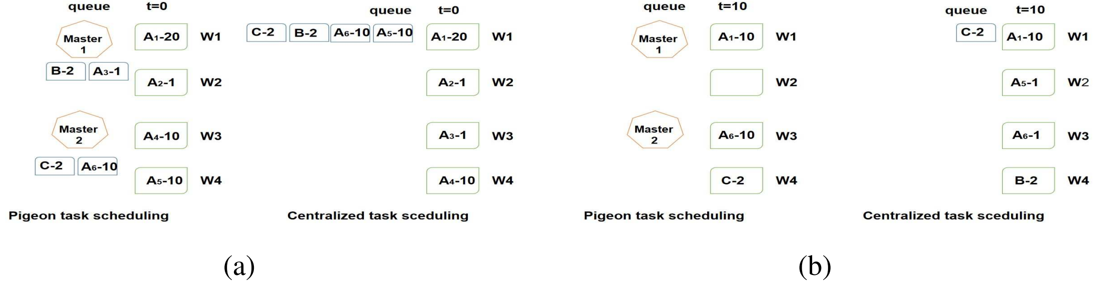


Figure 2.6: Task scheduling example: (a) tasks at time 0; (b) tasks at time 10. X-t at a worker or a queue: X is the task name, and t is the remaining task execution time.

the centralized one in the entire group size range studied. All the three percentiles of long job completion time decrease as the group size increases. The above results indicate that Pigeon is not only more scalable but also performs better than its centralized counterpart for handling heterogeneous jobs, particularly at high cluster loads.

Although the optimal group size may be workload dependent (such as the ratio between the number of short and long jobs, the task execution time distribution, etc.), based on the above observation, which agree with the observation made for the other two traces, and consistent with the analytical results for jobs at small fanout (i.e., the multiplexing gain is small when the group size is over 100), we can safely conclude that in general, the performance is insensitive to the group size in a wide range, i.e., between 50 and 100. Hence it suffices to set group size anywhere in between 50 and 100 and we set the group size at 100 for all the cases studied in this section.

Explanations for the Counterintuitive Phenomenon: A key observation we make is that this phenomenon may occur when both job fanout degree and task execution time vary in a wide range, which is the case in practice (see Table 2.1) but not for the model in the previous section (that explains why we did not observe this phenomenon there). The best way to see why this is true is by example.

Consider job scheduling for a single type of jobs and a cluster of 4 workers. At time 0, all the workers are idle and job A with 6 tasks (called as tasks A_1, ..., A_6) arrives, with task execution times of 20, 1, 1, 10, 10 and 10 units. Immediately following it are two other jobs B and C, each having 1 task with execution time of 2 units. We further assume that there is no processing overhead and the communication time can be neglected. Now we compare the performance of a Pigeon scheduler and its centralized counterpart.

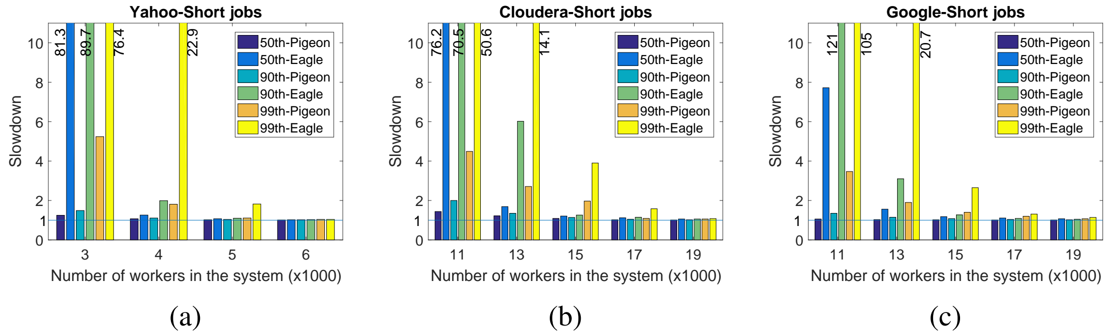
First consider a Pigeon scheduler, where 4 workers are divided into 2 groups with 2 workers each. Upon the arrival of jobs A, B, and C, in that order, the first 3 tasks from A

(i.e., A_1, A_2 and A_3 with execution times 20, 1 and 1) are sent to group one and the other 3 tasks from A (i.e., A_4, A_5 and A_6 all with execution time 10) to group two. Then the task from job B is sent to group one and the task from job C to group two. At time 0, in group one, two tasks A_1 and A_2 with execution times 20 and 1 are served by workers 1 and 2, respectively; and in group two, workers 3 and 4 serve tasks A_4 and A_5, respectively. The tasks at time 0 in Pigeon are shown in Figure 3.10 (a). At time 1, worker 2 completes the task A_2 and immediately starts serving the task A_3. It finishes the task A_3 at time 2 and then serves the task from job B which is completed at time 4. Hence job B is finished at time 4. At time 10, workers 4 and 5 complete the tasks A_4 and A_5, and then serve the tasks A_6 and C. The tasks in Pigeon are now given in Figure 3.10 (b). The task from job C is finished at time 12, so job C is completed at time 12. As tasks A_1 and A_6 are finished at time 20, so job A finishes at time 20. The job completion times in Pigeon for the three jobs are 20, 4 and 12, for a total of 36 units.

Now, consider a centralized scheduler. The first 4 tasks (i.e., A_1, A_2, A_3 and A_4) from job A are sent to workers 1-4 at time 0, respectively, as given in Figure 3.10 (a). At time 1, workers 2 and 3 complete their tasks and then serve the other two tasks (i.e., A_5 and A_6) from job A. At time 10, worker 4 finishes its task and then serves the task from job B which will be completed at time 12. So job B is completed at time 12. The tasks at time 10 is given in Figure 3.10 (b). At time 11, workers 2 and 3 complete their tasks, and then worker 2 serves the task from job C which is completed at time 13, and hence job C is completed at time 13. Task A_1 finishes at time 20, and hence job A finishes at time 20. So the job completion times for the three jobs are 20, 12 and 13, respectively, for a total of 45 units, 9 units or 25% more than the Pigeon scheduler!

From the above example, we see that for centralized scheduling, a job with a large fanout degree (i.e., job A) causes head-of-line blocking of the following jobs of the same type, even when their fanout degrees are low (i.e., jobs B and C). The head-of-line blocking caused by the same type of jobs may be alleviated by enabling task preemption [3,8], which however, may incur significant preempting overhead, particularly for resource-intensive tasks.

In contrast, for Pigeon, the tasks for jobs are distributed to different groups. This enhances the chance for tasks from later jobs to be served before tasks from the earlier jobs due to heterogeneous task execution time distribution. This helps reduce the chance of head-of-line blocking of jobs with small fanout degrees by jobs with large fanout degrees, hence, resulting in better overall performance. While helping more in alleviating head-of-line blocking by dispersing the tasks of a job with a large fan-out degree to more groups, using a smaller group size reduces multiplexing gain. This help explain why Pigeon gives the overall best performance at the group size in a certain range, i.e., 50 to 100.



(a) (b) (c)

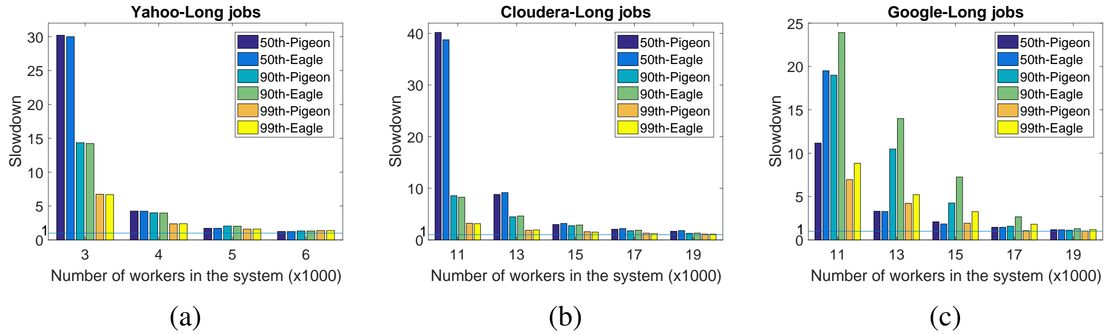
Figure 2.7: Short job completion time, $W = 20$.

Master Workload Estimation: Finally, with the parameters given above, we can now estimate the offered task load at a master. Assume that the cluster size, N_c=20,000, and hence, the total number of masters, N_g=200, given the group size, N_w=100. The real trace statistics in Table 2.1 suggest that the average task execution time is more than 100s (from 118s to 661s, to be exact). It means that a master needs to handle about only 1 task per second on average (or equivalently, 1 task per 100 seconds per worker), this overhead is negligible. In the case of a long job with a huge number of tasks, such as a job with 50,000 tasks, each master will see a burst of task arrivals of size 250. This is in stark contrast with a distributed scheduler, who needs to generate and dispatch 50,000 tasks. This example clearly indicates that the resource demand on a master is modest and a single worker should be sufficient to serve as a master, which consumes only 1% (i.e., 1 out of 100) of the total worker resources in the cluster. This means that indeed, Pigeon is a highly scalable solution.

In practice, to save the cluster resource, a master may run in a regular worker as long as the worker has enough resource to act as both a master and a regular worker. An alternative is to allow a worker to run multiple masters. For example, consider a system with 10,000 workers and each group with 100 workers. We may use 10 workers, each hosting 10 masters, instead of 100 workers with one master each, hence, cutting the overhead from 1% to 0.1%.

2.4.2 Performance evaluation

The number of workers in the whole cluster is used as a tunable parameter to adjust the load level. We use 50th, 90th and 99th percentile job slowdowns with respect to the case of *unlimited resources* (i.e., the case with zero communication time and zero task queuing time) for both short and long jobs as performance metrics. More specifically,



(a) (b) (c)

Figure 2.8: Long job completion time, $W = 20$.

the xth-percentile short/long job slowdown is defined as the xth-percentile short/long job completion time divided by the xth-percentile short/long job execution time. Here a job execution time is defined as the largest task execution time among all the tasks in the job.

figs/pigeon 2.7 and 2.8 give the slowdowns of the 50th, 90th and 99th percentiles of short and long jobs for all the three traces. Here W is set at 20 in Pigeon. First, we note that at the fixed job arrival rate, as the number of workers in the cluster increases, the slowdowns of the two schedulers converge and approach 1 for both short and long jobs. This is expected, because as the cluster size becomes larger, or equivalently, the load becomes lighter, all the jobs experience smaller queuing delays and hence, smaller job completion times, regardless what scheduling mechanism is used. Hence, it is more interesting and important to focus on small cluster sizes or high load regions. As the cluster size reduces, we can see that remarkable performance gaps between the two emerge.

In the case of the Yahoo trace, at the cluster size of 3K, the slowdowns for short jobs in Pigeon are about 1.3, 1.5 and 5.3 times which indicates the queuing times are less than one job execution time for the 50th and 90th percentiles, and just above 3 job execution times for the 99th percentile. The results indicate that Pigeon achieves excellent short job performance even at very high cluster loads (about 95%). In contrast, the slowdowns for Eagle are above 70 times for all the three percentiles, implying that for Eagle, the queuing times are more than 70 job execution times for short jobs. Similar results can be found with the Google and Cloudera traces as shown in figs/pigeon 2.7(b)-(c). In what follows, we explain why Pigeon outperforms Eagle by such big margins.

Eagle improves over Hawk, as detailed in [34], by allowing workers who are handling long jobs to reject the probes coming from distributed schedulers who handle short jobs. This allows a distributed job scheduler to issue more rounds of probes to discover workers that are not handling long jobs, hence alleviating the head-of-line blocking effect for short jobs. However, most lower priority (i.e., non-reserved) workers can still be

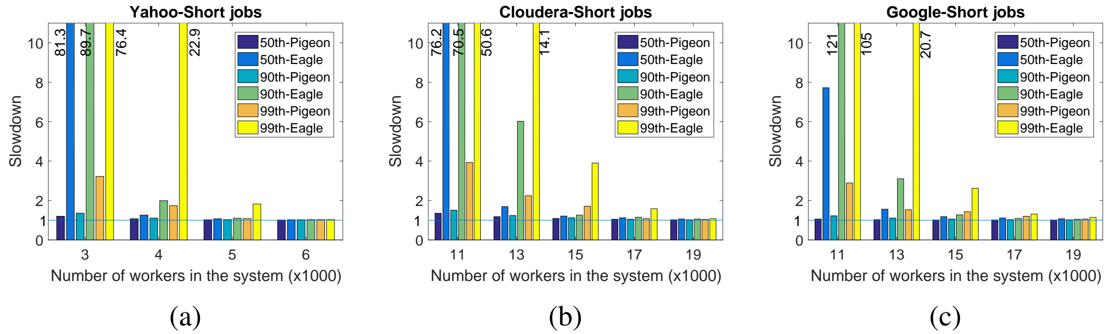


Figure 2.9: Short job completion time, $W = \infty$.

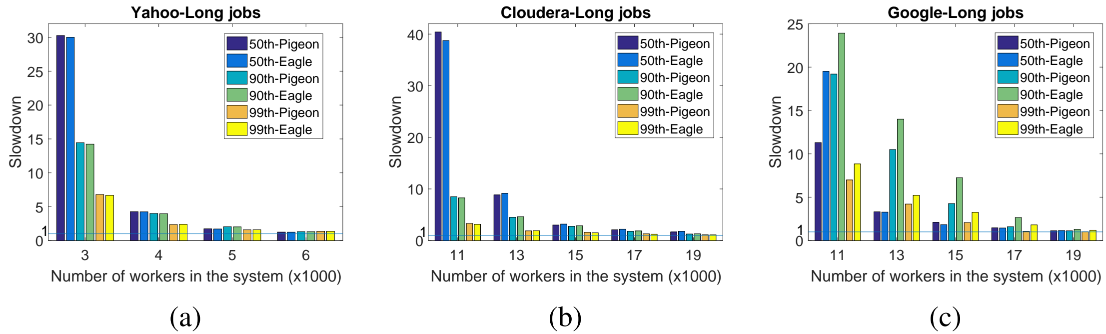


Figure 2.10: Long job completion time, $W = \infty$.

blocked by the long jobs, either at high load or whenever a long job with a large fan-out degree arrives. In this case, after multiple rounds of random probing, most of the tasks from short jobs are forced to be served by the high priority (i.e., reserved) workers, which however, may become the bottlenecks themselves. For example, for the Yahoo trace, consider the case of a cluster with 3K workers and 60 high priority workers (2% as set in Eagle [34]) for short jobs. When a long job with 5900 tasks (i.e., the maximum number of tasks in a job for the Yahoo trace) arrives, each low priority worker has to serve, on average, about 2 tasks of the job. After the tasks of the long job are placed, all the upcoming short jobs following this long job are forced to be served by only 60 high priority workers after a number of rounds of probing. In other words, all the low priority workers are blocked by the long job, hence resulting in big job completion time for short jobs. The key difficulty is that as a hybrid scheduler, Eagle distributes tasks from short and long jobs independently by distributed and centralized schedulers, respectively.

In contrast, Pigeon allows centralized scheduling of tasks coming from both short and long jobs and full resource sharing at the group level. This makes it possible for Pigeon to largely remove head-of-line blocking without starving the long job through weighted

fair queuing and worker reservation. Again, consider the above example where a long job with 5900 tasks arrives at a cluster with 3K workers. Assume that the workers are divided into 30 groups of 100 each with 2 (i.e., 2%) workers reserved for the tasks from short jobs. Now about 197 (i.e., 5900/30) tasks from the long job are sent to each group. In a given group, the master dispatches as many tasks out of 197 to the available low priority workers as possible and the rest to the low priority queue, e.g., with 10 to the available low priority workers at the load of 90% (i.e., about 90% or 88 out of 98 are currently busy) and 187 queued. The upcoming tasks of short jobs are either served by an idle reserved worker or queued in the high priority queue. However, in addition to the 2 high priority workers, whenever a low priority worker becomes idle, it will first have high chance (19/20 at $W = 20$) to serve a task from the high priority queue. Unlike Eagle, most of the long tasks (i.e., 187) are not queued at the low priority workers, but centrally at the master, high priority tasks following these low priority tasks will not be blocked by the latter from accessing the low priority workers. Moreover, a task at the head of the high priority queue is likely to find an idle low priority worker soon, because the probability that one out of 98 busy lower priority workers will finish its task in the near future is high. This explains why Pigeon can significantly outperform Eagle in terms of short job performance, especially in the high load region.

The fact that Pigeon performs slightly better than Eagle even for long jobs, despite the use of the weighted fair queuing for short jobs over long ones, as depicted in Figure 2.8, can be explained as follows. First, Pigeon generally reserves a smaller number of workers for short jobs than Eagle (i.e., 9% vs. 17% and 7% vs. 9% in the cases of the Google and Cloudera traces, respectively and 2% vs. 2% in the case of the Yahoo trace), hence allowing more workers to be used by the long jobs. This explains why overall Pigeon outperforms Eagle in the cases of the Google and Cloudera traces but not as much in the Yahoo trace. Second, for all the real traces studied, the overall execution time for short jobs constitutes less than 20% of the total job execution time, implying that the possible negative impact of giving high priority to short jobs (i.e., letting W=20) on the performance of long jobs is quite limited.

We also test the effect of W by comparing the setting of $W = \infty$ against that of W=20. The results are given in figs/pigeon 2.9 and 2.10. We can see that only the short job completion times at very high load are different. For example, when W changes from 20 to ∞, the slowdowns of the 50th, 90th, and 99th short job completion times for the Yahoo trace are reduced from 1.3, 1.5 and 5.3 to 1.2, 1.4 and 3.6, respectively, in a cluster with 3K workers. while the corresponding slowdowns for long jobs are within 2%. The results indicate that the performance of Pigeon is indeed insensitive to W, in the range of [20, ∞].

The above results clearly demonstrate that Pigeon is a much more effective job scheduler than Eagle in terms of both design complexity (e.g., without probing phase, without having to run two different types of schedulers, and no worker involvement of scheduling) and performance.

2.5 Performance Evaluation on EC2 Cloud

In this study, we compare the performance of Pigeon against both Sparrow [91] and Eagle [34], the state-of-the-art distributed and hybrid job schedulers, respectively, in a small cluster on the Amazon EC2 cloud. The Pigeon implementation includes two parts: the Pigeon scheduler code and the Spark plug-in. Distributed Pigeon schedulers are concurrently deployed at the application frontends, exposing services to allow the framework to submit job scheduling requests using remote procedure calls (RPCs). All RPCs for internal communications between modules of a Pigeon scheduler are defined with Apache Thrift [120]. We directly run the available open source implementation codes for Sparrow [91] and Eagle [34]. m4.large instances are used to serve as workers, masters and schedulers.

The cluster is composed of 10 schedulers and 120 workers. For Pigeon and Eagle, 10% of the workers are reserved for short jobs. In Pigeon, the workers are divided into 3 groups with 40 workers each. One worker in each group is selected as a master and W is set to infinite (i.e., each master runs two strict priority task queues). A sample job trace including 5000 jobs is extracted from the Google trace. The task execution time is scaled to the range of 10ms to 100s and the job fanout degree is scaled to the range of 1 to 100. The short job cutoff time is set at 1s. It turns out that 10% jobs are long jobs, which however, consume about 88% overall task execution time, in line with the statistics of the original trace.

We use average job arrival rate as a tuning knob to adjust the cluster load. As the Poisson arrival process has been widely considered a good model for datacenter workload, we assume that the job arrival process follows the Poisson distribution. The experimental results are also compared against the simulation results. The simulators for Pigeon and Eagle are the same as the ones described in the previous section and the open source event-driven simulator for Sparrow [91] is used.

We find that the short job performance for Sparrow and Eagle are very sensitive to the number of schedulers in use (by changing the number of schedulers from 1 to 10). This is because the processing delay in the probe phase becomes non-negligible compared to the job execution time for short jobs. In contrast, Pigeon offers almost the same performance, regardless how many schedulers are used. In all the experiments, we use 10 schedulers to minimize the impact of the processing delay for Sparrow and Eagle.

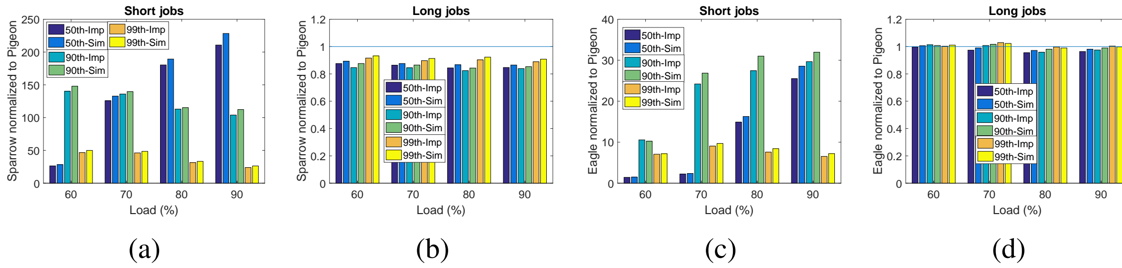


Figure 2.11: Experiment vs Simulation. Sparrow (a) short job and (b) long job; Eagle (c) short job and (d) long job.

Figure 3.9 depicts both measured (on EC2) (denoted as Imp) and simulated (denoted as Sim) 50th, 90th and 99th short and long job completion times normalized to Pigeon. The results for Sparrow and Eagle are depicted in figs/pigeon 3.9 (a) and (b) and figs/pigeon 3.9(c) and (d), respectively. Clearly, the experiment results are consistent with the simulation results. The differences between experiment and simulation are within 15% for short jobs and 5% for long jobs, mainly caused by the unaccounted processing overhead in the simulation.

As Sparrow does not distinguish between short and long jobs, it incurs up to 200 (10) times longer short job completion times than Pigeon (Eagle), although it offers up to 15% better long job completion times than both Pigeon and Eagle. This means that Sparrow is not effective in supporting short jobs in the presence of heterogeneous workloads. We also see that Pigeon provides significant performance gain for short jobs over Eagle. For example, at 90% load, the 50th, 90th and 99th percentile short job completion times for Eagle reaches about 25, 30 and 7 times longer than those for Pigeon. Pigeon and Eagle achieve comparable performance for long jobs at all cluster loads. The experiment results indicate that Pigeon is highly effective in handling heterogeneous jobs, which agrees with the simulation results obtained from the previous section.

The Pigeon project information and all the simulation and prototype implementation source codes can be found at https://github.com/ ruby-/pigeon.

2.6 Practical Considerations

This section discusses some practical implementation issues, i.e., how to handle master failure and how to deal with heterogeneous workers and task assignment constraints.

2.6.1 Master Failure Recovery

A master plays a key role in a group. If a master fails, all the group information, such as the queued tasks and idle worker lists, are lost. In a full-fledged implementation of Pigeon, one may borrow a failure recovery mechanism widely used in the traditional distributed systems for failure recovery [114]. To allow fast recovery from a master failure, a second master is selected in a group. The second master can be another worker. A master needs to periodically update the second master on the group information and task state information. Whenever a master failure is detected, the second master can immediately take the master responsibility from the failed master without losing any state information. After a master failure happens, the second master sends a notice to each worker in the group and each scheduler in the cluster to notify them of the changes, so that the subsequent tasks and idle worker notices are sent to the new master. Now the second master acts as a new primary master of the group and then a new secondary master should also be chosen for subsequent backups.

If both masters fail at the same time, the group information is lost. To quickly recover the group information, any worker in the group that detects such a failure can take the responsibility as a master. It broadcasts a message into the group to ask worker status. Each worker sends its response back to the new master with its status (idle or busy, priority, executing task, etc.). The new master also needs to send a message to each scheduler to get the task information sent to the group to recover the task queue list in the group. In case that multiple workers take the responsibility as a new master at the similar time, these workers can elect one as the new master based on some rules, e.g., the timestamp of master declaration time, CPU power or storage capacity and so on.

2.6.2 Dealing with Heterogeneous Workers and Tasks with Assignment Constraints

In the Pigeon design, we implicitly assumed that the same number of workers are assigned to each group and all the workers have the same processing power. In practice, however, the number of workers in a group may not be conveniently set to be the same. Even if the numbers of workers assigned to different groups are the same, different workers may have different processing powers. In this case, the schedulers in Pigeon may need to assign tasks to different groups in proportion to their relative processing powers to balance the task load among groups. More specifically, the probability of a task assigned to a group is proportional to the group's processing power.

Moreover, in practice, some tasks may have to be assigned to specific workers, as the needed resources or data are only available at those workers. All these may cause load

imbalance among worker groups and hence have a negative impact on the performance of Pigeon. One possible solution is to require that all masters report their queue lengths for all the priority queues periodically to distributed schedulers. This will allow distributed schedulers to make more informed decision as to how to balance the load among groups. The well-balanced task assignment will reduce the overall job completion latency and increase the overall throughput, and hence resulting in high system utilization.

2.7 Related Work

Today's datacenter job schedulers can be classified into three categories, i.e., centralized, distributed and hybrid. The earlier job schedulers, e.g., Jockey [44], Quincy [60], Tetrished [123], Delay Scheduling[139], Firmament [46] and Yarn [125] are centralized by design. A centralized scheduler can potentially provide high worker utilization, as it has a global view of the worker status for individual workers. But the scalability and head-of-line blocking are the major problems concerning centralized scheduling solutions. The scheduling decisions and status reports can overwhelm a centralized scheduler and cause additional job delay. Some shared-state schedulers, e.g., Apollo [17], Omega [103], and Mesos [56], use a centralized resource manager to maintain shared state. The job distributors are distributed but the decision making is based on the shared status of the cluster resource availability. The shared status is updated by the distributed schedulers and/or workers. However, the shared state may not be always up-to-date and hence may result in job placement conflict and retries. This approach still requires a central entity for shared status maintenance. Recent work BigC [22] and Karios [35] propose to deal with job heterogeneity by suspending long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks.

Yarn Federation [45] is developed to address the scalability issue of Yarn [125]. In Yarn Federation, a cluster is split into sub-clusters. Jobs are distributed to sub-clusters, Jobs are distributed to sub-clusters, each of which in turn performs job scheduling (i.e., distributing tasks of received jobs). With the coordination between resource managers and nodes from different sub-clusters, the tasks of a job can span the entire cluster, not limited to the sub-cluster the job is mapped to. As a result, YARN federation is more of a quasi-centralized task scheduling solution than a hierarchical one. Hydra [30] leverages the Yarn Federation architecture, in which a collection of loosely coupled sub-clusters coordinates to provide the illusion of a single massive cluster. In contrast, Pigeon is indeed a two-level hierarchical task scheduling solution, in which the tasks from a job spans across multiple groups (or sub-clusters). First, distributed job schedulers evenly distribute tasks of jobs to

all group masters. Then, in turn, each group master, which is job-agnostic, uses priority queues to differentiate the scheduling of short and long tasks. Moreover, while YARN federation aims to address the scalability issue of the resource manager in YARN, Pigeon mainly aims to address job heterogeneity concerning centralized and hybrid job scheduling solutions.

Sparrow [91], on the other hand, is a fully distributed job scheduler based on random batch-based probe and late task binding. Although free from the scalability issues that plague the centralized job schedulers, the distributed schedulers and workers in Sparrow need to maintain fairly large amounts of task related state information and incur high communication cost for probing, including probe management, probe queuing, probe processing, and redundant probe removals. Furthermore, it does not perform well in highly loaded clusters nor in the presence of heterogeneous workloads. Another probe-based distributed scheduler, Peacock [73], organizes workers in a ring overlay network and a probe can be rotated to its neighbors at fixed time intervals to balance the probe queue lengths among workers. Peacock, however, requires that the workers communicate with each other to form and maintain a ring topology. Moreover, it inherits much of the drawbacks pertaining to probe-based solutions in general.

To solve the scalability issue while providing high performance in the presence of heterogeneous jobs, Hybrid schedulers [36, 34, 69, 134] are proposed. Hybrid schedulers combine a centralized scheduler and a set of distributed schedulers. Mercury [69] uses distributed schedulers to place jobs without latency requirement and a centralized scheduler to place jobs with guaranteed resource requirement. Hawk [36] uses a centralized scheduler for long job placement and the distributed schedulers for short job placement. The short job scheduling is similar to the techniques used in Sparrow, i.e., batch probing and late task binding based. Some workers are reserved to serve short jobs only, as a way to mitigate head-of-line bocking. Moreover, an idle worker can steal tasks belonging to short jobs from other workers to improve efficiency. Eagle [34] improves over Hawk by introducing sticky batch probe with each probe staying on a worker until all the tasks of the job finish. It also allows multiple rounds of probing to mitigate head-of-line blocking. These hybrid schedulers need a central scheduler that can still pose a potential bottleneck. Moreover, short job scheduling is still probe-based and hence, inheriting its shortcomings.

More complex queuing mechanisms than priority queuing are being used to minimize the job performance. Queue reordering [34, 57, 123, 103] is used to reduce the job completion time. More complex queue management techniques [96] such as appropriate queue sizing, prioritization of task execution via queue reordering, and starvation freedom are also being used to improve the efficiency of job scheduling.

Some job scheduling solutions [67, 33, 116] are developed to improve the service level objectives (SLOs) violations. Morpheus [67] is designed to reduce the SLOs violations through automatically deriving SLOs and job resource models from historical data, relying on recurrent reservations and packing algorithms to enforce SLOs, and dynamic re-provisioning to mitigate inherent execution variance. The tail-cutting techniques [33, 116] can help mitigate the impact of stragglers on the job tail-latency performance.

Pigeon differs from the existing solutions in two important aspects. First, it is a hierarchically distributed solution to avoid head-of-line block in centralized schedulers. Second, it is free of the probing phase, a technique shared by all the existing distributed and hybrid solutions.

2.8 Conclusions

In this paper, we propose Pigeon, a distributed hierarchical job scheduler for datacenters. In Pigeon, workers are divided into groups. Each group has a master worker which centrally manages all the tasks handled by the group. Weighted fair queuing is used to provide priority service differentiation between tasks of short jobs and tasks of long jobs. A small portion of workers in each group are reserved to serve short job tasks only. The ability of each master in managing its group resources centrally makes Pigeon highly effective in scheduling heterogeneous jobs. The analysis, simulation and experiment results demonstrate that Pigeon outperforms Sparrow and Eagle by significant margins. Pigeon is implemented and tested in Amazon EC2 cloud, which has validated the Pigeon simulator used for the Pigeon evaluation.

2.9 Acknowledgments

We would like to thank our shepherd, Ahmed Eldawy, and the anonymous reviewers for their insightful feedbacks. This work is supported by the US NSF under Grant No. CCF XPS-1629625 and CCF SHF-1704504

CHAPTER 3

TailGuard: Tail Latency SLO Guaranteed Task Scheduling for Data-Intensive User-Facing Applications

A primary design objective for Data-intensive User-facing (DU) services for cloud and edge computing is to maximize query throughput, while meeting query tail latency Service Level Objectives (SLOs) for individual queries. Unfortunately, the existing solutions fall short of achieving this design objective, which we argue, is largely attributed to the fact that they fail to take the query fanout explicitly into account. In this paper, we propose TailGuard based on a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing policy (TF-EDFQ) for task queuing at individual task servers the query tasks are fanned out to. With the task queuing deadline for each task being derived based on both query tail latency SLO and query fanout, TailGuard takes an important first step towards achieving the design objective. A query admission control scheme is also developed to provide tail latency SLO guarantee in the presence of resource shortages. TailGuard is evaluated against First-In-First-Out (FIFO) task queuing, task PRIority Queuing (PRIQ) and Tail-latency-SLO-aware EDFQ (T-EDFQ) policies by both simulation and testing in the Amazon EC2 cloud. It is driven by three types of applications in the Tailbench benchmark suite. The results demonstrate that TailGuard can improve resource utilization by up to 80%, while meeting the targeted tail latency SLOs, as compared with the other three policies. TailGuard is also implemented and tested in a highly heterogeneous Sensing-as-a-Service (SaS) testbed for a data sensing service, with test results in line with the other ones.

3.1 Introduction

It has been widely recognized that the query tail latency for Data-intensive User-facing (DU) services, such as web searching, online social networking, and emergency response through edge-based crowdsensing, has a great impact on user experience and hence, business revenues. For example, for Amazon online web services, every 100-millisecond addition of query tail latency causes 1% decrease in sale [7]. To meet strict tail latency Service Level Objectives (SLOs), the resources for DU services are generally over-provisioned [26, 12], at the cost of reduced profit. As a result, a key design objective of a DU service,

called **the design objective** in short hereafter, *is to maximize the resource utilization or query throughput, while meeting tail latency SLOs for individual queries*.

However, achieving the above design objective is by no means easy. A query for a typical DU service may spawn a number of tasks, known as **query fanout**, to be dispatched to, queued and serviced in parallel in different servers or edge nodes where the data shards reside and the slowest task of the query determines the query response time [27, 33]. The range of query fanouts may differ from one service to another, e.g., up to several hundreds for online social networking [89], on the order of several thousands to tens of thousands for web search [33], and potentially up to millions for emergency response through edge crowdsening [99]. A small number of outliers (caused by, e.g., skewed workloads [116] or software/hardware resource variations [77]) can significantly impact the query tail latency performance [33]. While a large body of works have been devoted to alleviating the impact of outliers on the query tail latency performance (e.g., [97, 127, 93, 80, 38, 37, 136, 63, 51, 135, 112, 53]), to the best of our knowledge, no existing solution attempts to meet more than one query tail latency SLO to satisfy different performance requirements of individual users, while maximizing the resource utilization or query throughput, hence falling short of the design objective.

In this paper, we claim that a solution that stands a chance to achieve the design objective must be not only tail latency SLO aware but also query fanout aware. This is simply because *to meet a given tail latency SLO, the task resource demands for tasks belonging to queries with different fanouts are different*. For example, assume that with a given amount of resource allocated to process each task and the task response time for each task has 1% probability to be over 100 ms. Then the query response time for a query with fanout k_f has probability, 1-0.99^{k_f}, to be over 100 ms, meaning that a query with k_f=1 and k_f=100 have 1% and 63.4% probabilities of being over 100 ms, respectively. This implies that while a query with k_f=1 can meet the tail latency SLO in terms of the 99th percentile tail latency of 100 ms, a query with k_f=100 cannot. In order to allow the query with k_f=100 to also meet the same tail latency SLO, a task associated with the query must be allocated a much larger amount of resource so that the chance it will exceed 100 ms is as small as 0.01%. This ensures that the probably that the query response time exceeds 100 ms is 1-0.9999^{100} = 0.01 or 1%, i.e., meeting the same tail latency SLO as the query with k_f=1. This example clearly demonstrates that to meet a query tail latency SLO for all queries regardless query fanouts, the task resource demands for tasks belonging to queries with different fanouts are different and a task belonging to a query with a larger fanout demands more resources, confirming our claim.

The implication of the above observation is significant. First, even with all the queries sharing a given tail latency SLO, the tasks belonging to queries with different

fanouts should be treated differently, e.g., by being allocated different amounts of resource to closely match their resource demands so that all the queries can meet the tail latency SLO at the lowest possible resource consumption. Any solution that fails to take the query fanout explicitly into account is guaranteed to result in resource overprovisioning, simply because such a solution will have to allocate task resources based on the worst-case task resource demand. This partially explains why the way to meet stringent tail latency SLOs for large-scale DU services in today's datacenters is normally through resource overprovisioning [26, 12]. Our simulation results (see Section 4.2 for details) indicate that by taking fanout into account, TailGuard can improve resource utilization by 80% compared to the First-In-First-Out (FIFO) queuing policy, while meeting a stringent query tail latency SLO for DU workloads.

Second, consider a DU service that supports multiple classes of queries with a higher class requiring a more stringent tail latency SLO. Since the resource demand for a task is a function of not only the tail latency SLO but also the fanout of the query the task belongs to, it becomes apparent that a task associated with a query of a lower class but with a larger fanout may end up demanding more resources than a task in a query of a higher class but with a smaller fanout. This renders class-based task queue scheduling disciplines (e.g., PRIority-based task Queuing (PRIQ) [122, 48, 26]), task fanout-unaware queue management policies (e.g., the Tail-latency-SLO-aware Earliest-Deadline-First Queuing (T-EDFQ)), or task preemption [22] policies inadequate to achieve the design objective. This may also render some task reordering solutions solely based on task sizes [77, 86] inadequate. Our simulation results (see details in Section 4.2) demonstrate that TailGuard can improve overall resource utilization by 40% over the PRIQ policy and 22% over T-EDFQ in supporting two classes of tail latency SLOs for DU workloads.

In this paper, we propose TailGuard, a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing(TF-EDFQ) policy, as a first step towards achieving the design objective for DU services in general. As a top-down approach, TailGuard decouples the upper query level design from the lower task level design. First, at the query level, a task decomposition technique is developed to translate the query tail latency SLO for a query with a given fanout into a task queuing deadline for tasks spawned by the query at the task level, reflecting the resource demand of the tasks. This effectively decomposes a hard cotask scheduling problem at the query level into individual queue management subproblems at the task level. Second, at the task level, a single TF-EDFQ corresponding to a task server is used to enforce the task queuing deadlines, as a way to differentiate resource allocation for tasks with different resource demands. In principle, TailGuard permits unlimited number of query classes and is lightweight, as it incurs minimum overhead for task queuing deadline estimation and requires to implement only a single earliest-deadline-first queue per task

server for any DU applications. A query admission control scheme is also developed to provide tail latency SLO guarantee in the face of resource shortages.

TailGuard, or equivalently, TF-EDFQ, is evaluated against FIFO, PRIQ and T-EDFQ (Section 3.1 gives their exact definitions) by both simulation and testing in the Amazon EC2 cloud. Three traces generated from the Tailbench benchmark suite [71] are used as input. The results demonstrate that TailGuard can improve resource utilization by up to 80%, while meeting the targeted tail latency SLOs, as compared with the other three policies. The query admission control scheme is also tested and the results indicate that it can indeed provide query tail latency SLO guarantee. Finally, TailGuard is implemented and tested in a highly heterogeneous Sensing-*as*-a-Service (SaS) testbed for an edge-based temperature-and-humidity sensing service, with test results in lines with the other ones.

3.2 Background and Related Work

3.2.1 Data-Intensive User-Facing Services

DU services are a predominant class of workloads in today's cloud and have also emerged as an important class of workloads in an edge-cloud ecosystem, generally known as SaS[1][92, 108]. Predominant DU services are driven by queries that require query responsiveness in sub-seconds to seconds and may need to touch on massive datasets, which are typically carried out in a data parallel fashion. The working dataset for a service (e.g., the total amount of crowdsensing data in the case of an SaS) in this class are distributed to a large number of task servers/edge nodes. Accordingly, a query may spawn a number of tasks to be dispatched to some or all of these task servers/edge nodes to be processed. A notable subclass of such services is OnLine Data-intensive (OLDI) services [83]. A query for an OLDI service needs to touch upon every part of the working dataset, i.e., the query fanout for each query is equal to the total number of servers involved (ranging from a few to tens of thousands). Large online search products, online advertising and online machine translation, are examples of OLDI services. For other DU services, different queries may need to touch upon different parts of the working dataset. A notable example of such a service is social networking services, such as Facebook and LinkedIn. For instance, the fanout for a typical Facebook page query is in the range of one to several hundreds with 65% under 20 [89]. Other examples are emergency response SaSes, e.g., finding a missing person through surveillance cameras and fire detection and alert via crowd temperature

[1]For an SaS, users send sensing requests to the cloud. The cloud then dispatches related query tasks to geo-distributed edge nodes to acquire desired sensing data collected and processed through crowdsensing, which are subsequently merged in and returned to the users from the cloud.

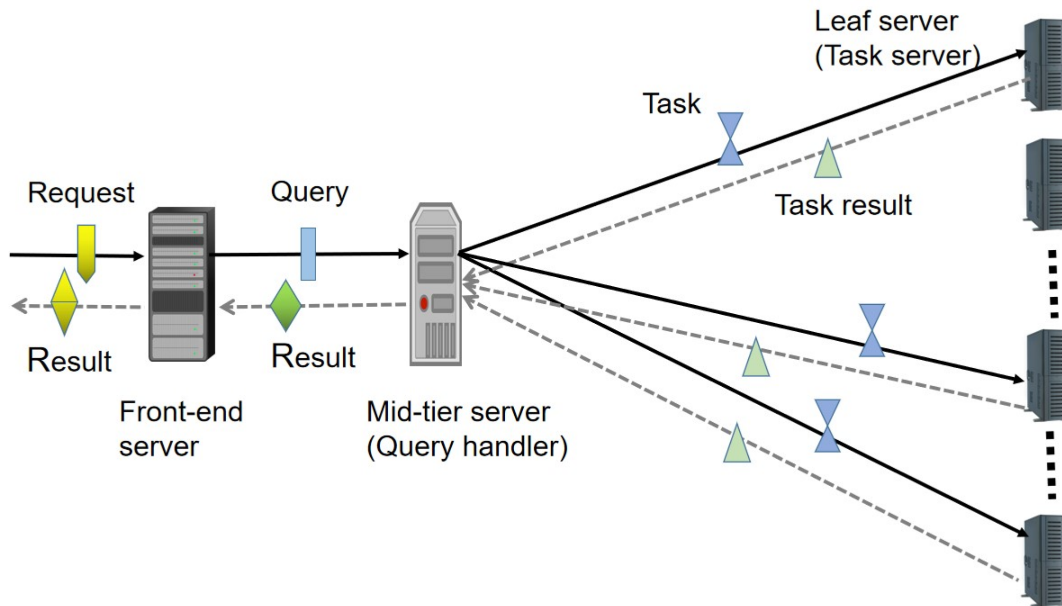


Figure 3.1: A typical DU application process architecture

sensing. A query of such a service is expected to have a fanout anywhere between one to a few millions depending on the scope of sensing.

A DU service may be launched in a dedicated datacenter cluster owned by a service provider, e.g., the web search service by Google, in a cloud by a tenant who rents cloud resources from a cloud service provider (e.g., Amazon cloud), or in an edge-cloud ecosystem owned by multiple stake-holders, including individuals who own the sensing data and/or edge devices and cloud service providers.

Figure 3.1 depicts a generic DU application processing model [111, 83]. It is composed of three parts, including a front-end server, a mid-tier server (called query handler in this paper), and a set of back-end leaf servers (called task servers in this paper[2]), each hosting a piece of the total dataset, also known as a shard, a partition, or a published sensing dataset (e.g., in an edge node).

When a user request arrives at the front-end server, its workflow is parsed to generate a set of queries to be issued sequentially to the query handler at the mid-tier server. Due to query/task dependency, the next query cannot be issued until the current one finishes. For each query received, the query handler spawns a number of tasks for the query and

[2]Task servers are also known as, e.g., workers, virtual-machines (VMs), containers, or edge nodes, depending on the specific services to be studied.

dispatches them to the queues corresponding to the task servers[3] that will serve them when they reach the queue heads. The tasks for the same task server are queued based on a given queuing mechanism. In practice, task servers are usually allocated dedicated CPU/memory/storage resources in the form of, e.g., cores, VMs, containers, or pods, as well as fix-sized data shards, forming a more or less homogeneous task server cluster. As a result, the differentiation of resource allocation among tasks with different resource demands are mainly through task queuing policies, e.g., PRIQ [122, 48, 26], task-reordering-based queuing [77, 86, 94], or EDFQ, unless task-aware resource auto-scaling [101] is allowed.

Upon completion of the execution of a task, the task result is returned to the query handler to be merged with the task results from the other tasks of the query. The query finishes when all the task results are merged and sent to the front-end server. Hence the task response time for the slowest task dictates the query response time. In turn, the request completes when the last query in the request finishes.

3.2.2 Tail Latency Aware Solutions for DU Services

Many works have been devoted to addressing query tail latency related issues for DU services, which can be broadly classified into two categories, i.e., *outlier alleviation*, focusing on curtailing the tail length of the task response time to improve overall query tail latency performance, and *tail latency SLO guarantee* for queries sharing a single tail latency SLO. In what follows, we elaborate more on the solutions in the two categories, respectively.

Outlier Alleviation: Most existing solutions fall into this category. Some typical examples in this category are listed as follows. Solutions based on task-size-aware task reordering in a task queue [77, 86, 94**?**] are proposed to avoid head-of-line blocking of small-sized tasks by large-sized ones to reduce the mean task latency. Task-aware scheduling schemes [93, 80, 38, 37, 136**?**] are designed to shorten the tail latency for tail latency critical tasks in workloads with both batch and tail latency critical queries. Redundant-task-issue solutions [127, 65, 116, 112] are developed to reduce the task tail latency by allowing a task to be issued to multiple task server replicas. Task execution time prediction through workload profiling [97, 63, 51, 135**?**] and machine learning [62, 90, 46] are widely employed to adjust the level of parallelism to remove task bottlenecks or to avoid sending tasks with predicted long execution time to poorly performing task severs to reduce task tail latency. Solutions based on synchronized garbage collection for all task servers [115, 33] are proposed to minimize variabilities of task execution times among parallel tasks to reduce query tail latency. Solutions that allow partial results to be returned to fulfill a query,

[3]Note that the queuing may take place either centrally at the query handler or at individual task servers.

e.g., [53], can maintain more predictable query tail latency at the cost of possible loss of partial results. Dynamic resource allocation based on the feedback loop control mechanisms [15, 77, 20] are proposed to help reduce query tail latencies. CPU power control schemes [52, 68, 70, 83, 112**?**] are developed to dynamically adjust voltage and frequency scaling (DVFS) for task servers based on task execution time to save energy and maintain low task tail latency. A query fanout control scheme [27] is designed to control the fanout in queries to optimize the system performance. Flow deadline-aware datacenter networking solutions [124, 132, 9, 75, 47] and coordinated coflow scheduling in datacenter switches [119, 28, 8, 39] are also developed to provide predictable delays for task related flows and hence reduce query tail latencies, usually at the cost of significant software/hardware changes to datacenter switches. A transaction scheduling solution for geo-distributed databases [23] uses transaction timestamps to reduce both mean and tail latencies for edge computing. All these solutions help reduce the query tail latency, but cannot provide SLO guarantee.

Tail Latency SLO guarantee: There are a few existing solutions in this category, including Cake [128], PriorityMeister [143], SNC-Meister [141], WorkloadCompactor [142] and PSLO [78], all for shared datacenter storage applications. All these solutions, except Cake, aim at meeting a single query tail latency SLO for all queries with fanout of one only. Cake can handle fanout of more than one, but is unable to enable per-class or per-query tail latency SLOs, as it relies on direct measurement of the overall tail latency statistics as input for control, resulting in fanout-unaware resource overprovisioning. Clearly, a solution based on direct tail latency statistics measurement like Cake cannot be extended to allow per-query resource allocation, simply because the needed statistics are unavailable at this granularity. Some tail latency SLO guaranteed solutions for micro-service such as GrandSLAm [**?**] and Sinan [**?**] are proposed. But, again, they cannot support per-query tail latency SLO.

3.3 TailGuard

In this section, we first give the TailGuard query processing model. Then we present the task decomposition, or equivalently, task queuing deadline estimation solution and address its implementation issues. Finally we present the query admission control scheme. The major symbols used in TailGuard are listed and defined in Table 3.1.

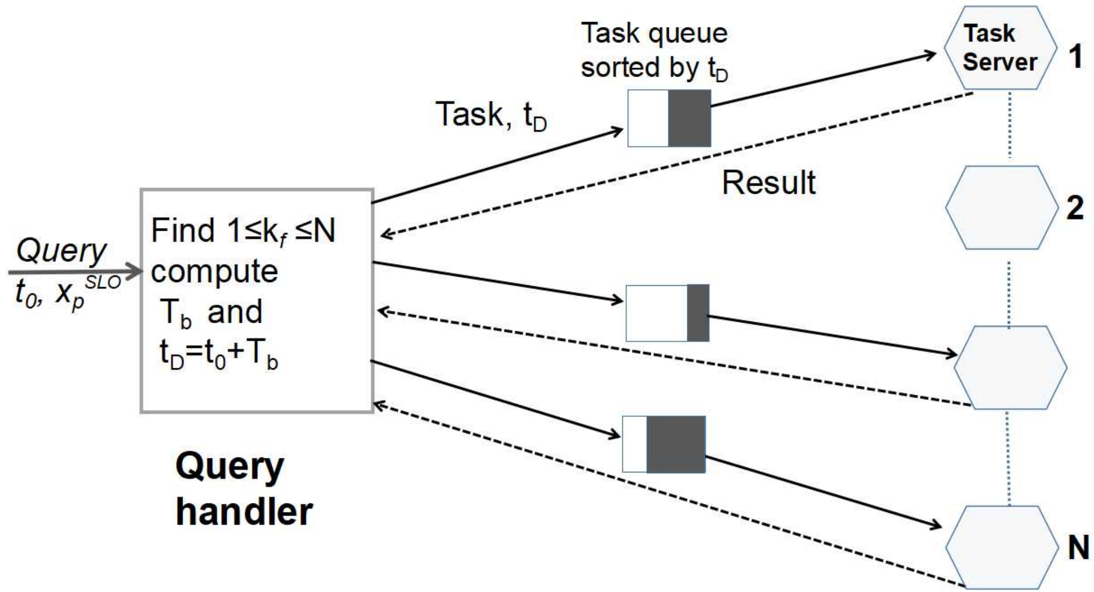


Figure 3.2: TailGuard query processing model. A task queue for a task server can be set in the task server or in the query handler.

3.3.1 TailGuard Query Processing Model

Consider a query processing model directly derived from Figure 3.1, as depicted in Figure 3.2. It is composed of a query arrival process, a query handler, and N task servers. The query arrival process characterizes the randomness of queries arriving at the query handler.

At the query level, upon receiving a query at time, t_0, the query handler first determines how many tasks (i.e., the query fanout, k_f) need to be spawned and to which k_f task servers these tasks need to be dispatched. The query handler estimates task pre-dequeuing time budget T_b and computes the task queuing deadline $t_D = t_0 + T_b$, shared by all the tasks associated with the query [4]. Here t_D is defined as the deadline when the task must be dequeued and given to the corresponding task server to be processed in order to meet the tail latency SLO for the query. As we shall show in the next subsection, T_b (or t_D) is a function of both query tail latency SLO in terms of the pth percentile query latency of x_p^{SLO} and query fanout, k_f, i.e., $T_b = T_b(x_p^{SLO}, k_f)$ and $t_D = t_D(x_p^{SLO}, k_f)$. Finally, the

[4]The rationale for assigning the same budget to all the tasks of a query is as follows. Mathematically, with two reasonable assumptions made, i.e., a task resource demand is an decreasing function of the task budget and the sum of the task budgets for all the tasks in a query must be upper bounded to meet a given query tail latency SLO, it can be easily shown that assigning the same budget results in the minimum overall resource allocation.

Symbol	Description
N	number of task servers
k_f	fanout of a query
T_b	task pre-dequeuing time budget for a query
t_0	query arrival time
t_D	task queuing deadline, $t_D = t_0 + T_b$
t_{pr}	task pre-dequeuing time
t_{po}	task post-queuing time or unloaded task response time
t_r	task response time, $t_r = t_{pr} + t_{po}$
x_p^{SLO}	pth percentile query tail latency SLO
$x_p^u(k_f)/x_p(k_f)$	unloaded/loaded pth percentile tail latency for a query with fanout k_f
$F_l^u(t)/F_l(t)$	CDF of unloaded/loaded task response time with respect to task server l
$F_Q^u(t)/F_Q(t)$	CDF of unloaded/loaded response time for a query
$P(k_f)$	probability of a query with fanout k_f

Table 3.1: The symbols used in TailGuard.

tasks, together with their deadlines, are dispatched to the queues corresponding to the task servers. Since task pre-dequeuing time budget, T_b, is an explicit function of both x_p^{SLO} and k_f for the query, *TailGuard by design permits per-query tail latency SLOs*.

At the task level, each task queue adopts a TF-EDFQ, based on $t_D(x_p^{SLO}, k_f)$. When a task is to be enqueued at a task queue, if the corresponding task server is idle, the task is serviced immediately, otherwise, it is inserted into the task queue with tasks ordered in increasing order of t_D's, hence with the task of the smallest t_D at the head of the queue. Whenever a task in service finishes, the task at the head of the queue is put in service immediately. Finally, upon the completion of execution of a task, the task result is sent back to the query handler to be merged. A query finishes as soon as the merging of all the task results completes.

TailGuard ensures that tasks with a higher chance to cause the violation of the associated query tail latency SLO will be serviced earlier, thus improving the system utilization.

Finally, as mentioned in Section 1, the performance of TailGuard will be compared against FIFO, PRIQ and T-EDFQ. In terms of queuing policy, FIFO is simply a first-in-

first-out queuing policy. PRIQ assigns tasks of different classes to different queues with strict priorities given to the queue of a higher class over that of a lower class. T-EDFQ works the same way as TailGuard except that $t_D = t_0 + x_p^{SLO}$. In other words, the queuing deadline for a task is dependent on the corresponding query tail latency SLO, x_p^{SLO}, but independent of query fanout, k_f. Clearly, both PRIQ and T-EDFQ degenerate to FIFO if all queries have the same tail latency SLO, i.e., the case with a single class.

3.3.2 Task Queuing Deadline Estimation

The key to the design of TailGuard is the task queuing deadline estimation or task decomposition. In this subsection, we first present the task queuing deadline estimation solution and then propose a way to implement it.

3.3.2.1 Solution

The task queuing deadline estimation problem can be formally stated as follows: *For a query with fanout, k_f, a given tail latency SLO in term of x_p^{SLO}, and arrival time, t_0, find the task queuing deadline, $t_D = t_0 + T_b(x_p^{SLO}, k_f)$, for tasks spawned by the query.* Here, $T_b(x_p^{SLO}, k_f)$, the task pre-dequeuing time budget, is the maximum allowable task pre-dequeuing time before the task must be dequeued and given/sent to the task server to be processed, in order to meet the query tail latency SLO.

First, we note that the task response time (also called loaded task response time), t_r, can be generally expressed as, $t_r = t_{pr} + t_{po}$, where t_{pr} represents the task pre-dequeuing time and t_{po} stands for task post-queuing time or unloaded task response time. t_{pr} is composed of task scheduling time and task queuing time, if task queuing takes place centrally at the query handler. It also includes task dispatching time, if task queuing occurs at the task server. t_{po} includes all the times the task incurs after de-queuing.

Now we assume that the Cumulative Distribution Function (CDF) of the unloaded task response time t_{po}, $F_l^u(t)$, with respect to task server, l, can be measured and updated (see Section 3.2.2 for details) for all task servers $l = 1, ..., N$. Furthermore, let $x_p^u(k_f)$ and $F_Q^u(t, k_f)$ represent the pth percentile unloaded query tail latency for a query with fanout k_f and the CDF of unloaded query latency, respectively. Here, a query latency is considered as unloaded (loaded) if the query response time does not (does) include pre-dequeuing delay, t_{pr}. Also define $n = n(k)$ to be the mapping between the k-th task in a query and the n-th task server the task is dispatched to, for $k = 1, ..., k_f$. Clearly, the unloaded query latency is

the task post-queuing time of the slowest of all k_f tasks. According to the ordered statistics [3], we have,

$$F_Q^u(t, k_f) = \prod_{k=1}^{k_f} F_{n(k)}^u(t). \tag{3.1}$$

By definition, we have,

$$x_p^u(k_f) = F_Q^{u-1}(\frac{p}{100}), \tag{3.2}$$

where $F_Q^{u-1}(.)$ is the inverse function of $F_Q^u(.)$.

Assuming that all the tasks in a query experience the same pre-dequeuing delay t_{pr}, we can express the CDF of the response time for task l, $F_l(t)$, as follows,

$$F_l(t) = \begin{cases} F_l^u(t - t_{pr}), & if\ t \geq t_{pr} \\ 0, & otherwise. \end{cases} \tag{3.3}$$

Hence

$$F_Q(t, k_f) = \prod_{k=1}^{k_f} F_{n(k)}(t) = \begin{cases} F_Q^u(t - t_{pr}, k_f), & if\ \ t \geq t_{pr} \\ 0, & otherwise, \end{cases} \tag{3.4}$$

and

$$x_p(k_f) - t_{pr} = F_Q^{u-1}(\frac{p}{100}). \tag{3.5}$$

From Eqns. (3.2) and (3.5), we have,

$$x_p(k_f) = x_p^u(k_f) + t_{pr}. \tag{3.6}$$

This result means that with any given query tail latency SLO, x_p^{SLO}, as long as, $t_{pr} \leq x_p^{SLO} - x_p^u(k_f)$, the query tail latency SLO is guaranteed to be met, i.e., $x_p(k_f) = x_p^u(k_f) + t_{pr} \leq x_p^{SLO}$. This means that the task pre-dequeuing time budget $T_b(x_p^{SLO}, k_f)$ can be defined as, $T_b(x_p^{SLO}, k_f) = x_p^{SLO} - x_p^u(k_f)$, or equivalently, the task queuing deadline can be defined as,

$$t_D = t_0 + T_b(x_p^{SLO}, k_f) = t_0 + x_p^{SLO} - x_p^u(k_f). \tag{3.7}$$

In other words, for a query arrived at $t = t_0$, as shown in Figure 3.2, so long as all the tasks belonging to this query are dequeued no later than t_D, the query tail latency SLO, x_p^{SLO}, is guaranteed to be met.

Ideally, under the work conserving condition[5], if a queuing policy can ensure that all the tasks exactly meet their queuing deadlines, the design objective is achieved. In practice, however, such a queuing policy may not exist. As a first step, TailGuard adopts EDFQ based on t_D, i.e., TF-EDFQ, to enforce the task queuing deadlines. This queuing policy can ensure that the task with the earliest queuing deadline is placed at the head of the queue before deadline. However, it cannot guarantee that the task at the head of the queue can always be served before deadline, simply because the task ahead of it may be still in service when the deadline is reached. On the other hand, the task may also have a chance to be dequeued before deadline, if the task server becomes idle before deadline. This implies that TailGuard may tolerate a small percentage of tasks missing their deadlines without violating the tail latency SLOs as the tail latency is a probabilistic measure. This observation underlays the query admission control solution given in Section 3.3.

A remark on meeting request tail latency SLO: Here we present preliminary ideas on how to extend the above task decomposition technique for queries to a task decomposition technique for requests that account for query/task dependencies.

Consider a request composed of M queries to be issued sequentially and with the request tail latency SLO expressed in terms of the pth percentile of request latency of, $x_p^{R,SLO}$. Now, the request response time $t_r^R = \sum_{i=1}^{M} t_{r,i}$, where $t_{r,i}$ is the query response time for the i-th query. Although this relationship is an additive one, the one for the corresponding tail latency is not. As the CDF of the request response time, $F_R(t)$, is the convolutions of all the CDFs of the constituent query response times, in general, $x_p^{R,SLO} < \sum_{i=1}^{M} x_{p,i}^{SLO}$, making query decomposition for requests difficult. In what follows, we show that the above task decomposition technique can be generalized to establish an additive relationship between the request pre-dequeuing time budget and task pre-dequeuing time budgets for the constituent queries, paving the way for the development of a task decomposition technique for requests.

Define unloaded request latency, $t_{po}^R = \sum_{i=1}^{M} t_{po,i}$, and the CDF of the unloaded request response time, $F_R^u(t)$, to be the CDF of t_{po}^R, where $t_{po,i}$ is the unloaded query latency for the i-th query. Further assume that all the tasks of query i have the same pre-dequeuing time, $t_{pr,i}$, and define request pre-dequeuing time, $t_{pr}^R = \sum_{i=1}^{M} t_{pr,i}$. Then we have the loaded request response time $t_r^R = \sum_{i=1}^{M}(t_{po,i} + t_{pr,i}) = t_{po}^R + t_{pr}^R$. Clearly, by substituting t_r, t_{pr}, t_{po}, F_Q, and F_Q^u with t_r^R, t_{pr}^R, t_{po}^R, F_R, and F_R^u, respectively, and following Eqs. (3.5) and (3.6), we have,

$$x_p^R = x_p^{Ru} + t_{pr}^R = x_p^{Ru} + \sum_{i=1}^{M} t_{pr,i} \tag{3.8}$$

[5]The work conserving condition refers to the condition whereby the task server is always busy as long as there are unfinished tasks at the server.

where x_p^R and x_p^{Ru} are the loaded and unloaded pth percentile tail latency of the request. Eq. (3.8) means that the request pre-dequeuing time budget, $T_b^R = x_p^{R,SLO} - x_p^{Ru}$, and it is additive, i.e., $T_b^R = \sum_{i=1}^{M} T_{b,i}$, here $T_{b,i}$ is the task pre-dequeuing budget for query i, for $i = 1, ..., M$.

Note that as long as T_b^R (i.e., $t_{pr}^R \leq T_b^R$) is met, the request tail latency SLO will be met, regardless the assignments of $T_{b,i}$'s. However, different assignments may lead to different resource utilizations. Hence, a key challenge that will be the main focus of our future work is: with a given total budget T_b^R, how to assign budgets $T_{b,i}$ to individual queries so that the resource utilization is maximized.

A remark on the relationship with outlier-alleviation solutions: Finally, note that TailGuard is orthogonal and complementary to many of the existing outlier-alleviation solutions. While an outlier-alleviation solution can help effectively reduce the tail lengths of the task service time distributions, $F_l(t)$'s, hence, improving the achievable query tail latency SLOs, TailGuard ensures that the improved tail latency SLOs can indeed be achieved at the minimum resource consumption (see Section 4.5 for a case study).

3.3.2.2 Implementation

The above task queuing deadline estimation solution requires the availability of the task post-queuing time distributions, $F_l(t)$, for all the task servers, l=1,...N, which must be conveyed to the query handler for task pre-dequeuing time budget estimation. Here, we propose an approach to estimate $F_l(t)$'s by means of a combined initial offline estimation process and a periodical online updating process.

Offline Estimation Process: As mentioned earlier, DU services are likely to run in a more or less homogeneous cluster. So before the service starts, we set $F_l(t) \approx F(t)$, for l=1,...N. This lends us a handy way to perform an initial offline estimation of only a single distribution function $F(t)$, which serves as the initial distribution for all the task servers.

More specifically, use a query handler and single task server and load it with a typical task workload trace to collect a sufficient number of samples of task post-queuing times offline. Then use these samples to construct $F(t)$ to be used as the initial distribution function for all task servers. This will allow task queuing deadlines to be estimated at the very start of a DU service.

Online updating process: To account for the inevitable heterogeneity in practice (e.g., due to skewed workloads, uneven resource allocation and resource availability changes), $F_l(t)$'s must be periodically updated online. Fortunately, this can be done with low cost. When the query handler receives and merges the task result for a task from task server l, it uses the current time minus the task dequeue time (which is either locally available if the

queuing takes place in the query handler, or comes with the task result from the task server l) as the post-queuing time for the task to update $F_l(t)$. This updating process accounts for all the possible post-queuing delays incurred by the tasks, including the long delays caused by outliers. Hence TailGuard captures heterogeneity through online updating process.

TailGuard implementation complexity: The computation complexities for both task queuing deadline estimation and queuing management in TailGuard are low. The former entails the evaluation of two equations, i.e., Eq. (3.2) for $x_p^u(k_f)$, which can be done in the background for all possible k_f's in advance and updated when $F_l(t)$'s change and Eq. (3.7) for each query. The latter requires the management of a single EDFQ. As a result, TailGuard is a lightweight solution.

3.3.3 Query admission control

TailGuard can provide tail latency SLO guarantee for all queries, when there are enough resources to sustain the workload. In the presence of resource shortages due to, e.g., sudden surges of workloads or hardware/software failures, some upcoming queries may need to be rejected to ensure that all admitted queries can meet the prepaid tail latency SLOs. Query admission control is particularly desirable in the case where resource auto-scaling cannot be done, e.g., due to monetary budget or resource constraints (e.g., edge resources may be quite limited to allow an SaS to scale).

We tested TailGuard using various workloads and found that the query tail latency SLOs can still be met, when a small portion (less than 2% in our tests) of tasks miss their deadlines, confirming the aforementioned observation. With this understanding, TailGuard sets a threshold for the percentage of tasks missing their deadlines, R_{th}, for query admission control. If the task queuing takes place centrally at the query handler, the information on whether a task misses its deadline or not is immediately available to the query handler, otherwise, this information can be piggybacked on the task results returned from the task sever. The query handler can update the task deadline violation ratio in a given moving time window. When the ratio exceeds R_{th}, upcoming queries are rejected, till the ratio falls back below R_{th} again. The moving time window can be set to be the same as the time window in which the tail latency SLOs should be guaranteed.

3.4 Performance Evaluation

To cover a wide range of applications, TailGuard is firstly evaluated based on simulation using the workload statistics for three datacenter applications available in Tailbench [71] as input. We first characterize the workload and then present the simulation results

along the fanout and service class dimensions; with an outlier alleviation scheme; and with query admission control. We then implement and test TailGuard in the Amazon EC2 cloud. Finally we verify TailGuard in a highly heterogeneous SaS testbed.

3.4.1 Workloads

For simulation, a DU workload must be characterized by a query arrival process, a query fanout distribution and a task post-queuing time distribution. Unfortunately, the available real traces simply do not contain the needed information. Although traces for commercial DU services in cloud are available, e.g., those made available by Google [122, 102] and Alibaba [48, 81, 26, 121], they only include the CPU and memory usage information for task servers, not the information needed to drive the simulation at the task level, including the arrival process, query fanouts and task service times. Hence, we resort to modeling for the first two and benchmarks for the third one, as described in detail below.

First, since the Poisson process [5] has been widely recognized as a good model for cloud applications in general [83], by default, we assume that the query arrival process is Poisson with mean arrival rate, λ, a tunning knob to adjust the system load. Meanwhile, to test the performance sensitivity of TailGuard with respect to the burstiness of query arrivals, a burstier arrival process, i.e., the Pareto arrival process [4], is also used in one simulation case.

Second, although a few publications do offer fanout distribution, $P(k_f)$, for k_f=1,...,N, for the DU services, e.g., the Facebook social networking service [89], they do not provide task service times needed for the task-level simulation. This, however, should not be too much of a concern, as TailGuard needs to be applicable to both the existing and future workloads whose $P(k_f)$'s are not known yet. Hence, we adopt quite different $P(k_f)$ models for different case studies to gain a wide coverage. As we shall see, for all those cases tested, TailGuard consistently outperforms the FIFO, PRIQ and T-EDFQ queuing policies, which strongly suggests that the TailGuard's performance gain is insensitive to $P(k_f)$'s.

Third, as a solution meant to be used by the current and future DU services in general, TailGuard should be tested against DU services with a wide range of task service time distributions. To this end, we resort to Tailbench [71] to gain access to applications with a wide range of task service time distributions. Tailbench provides eight DU task benchmarks. Each of these workloads allows a sufficiently large number of task service time samples to be collected to construct $F(t)$ for task service time, assuming that the post-queuing time, t_{po}, is dominated by the task service time, for the lack of the information about the rest of the post-queuing delays. We further assume that $F_l(t)$=$F(t)$ for l=1,...,N,

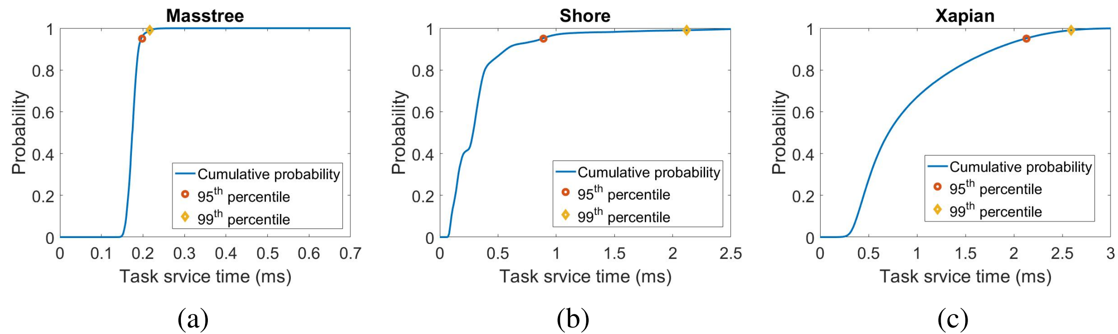


Figure 3.3: The CDFs and the unloaded 95th and 99th percentile task tail latencies of the three Tailbench workloads.

Bench	T_m	$x_{99}^u(1)$	$x_{99}^u(10)$	$x_{99}^u(100)$
Masstree	0.176	0.212	0.247	0.473
Shore	0.341	2.117	2.721	2.829
Xapian	0.925	2.592	2.998	3.307

Table 3.2: The mean task service time T_m (ms) and the unloaded 99th percentile query tail latency x_{99}^u (ms) with various fanouts.

i.e., the homogeneous case, which do not change over time (All the other delays and heterogeneity will be accounted for partially in the Amazon EC2 case study and fully in the SaS case study). These workloads can be classified into three groups with distinct characteristics for $F(t)$. We select one workload from each group to be tested, including Masstree for in-memory key-value store, Shore for SSD-based transactional database and Xapian for web search.

Figure 3.3 depicts the CDFs and the unloaded 95/99th percentile task tail latencies for the three workloads. Table 3.2 also gives the related statistics, including the mean task service time (T_m) and the unloaded 99th percentile query tail latency at fanouts k_f=1, 10 and 100, derived from Eqs. (3.1) and (3.2).

3.4.2 Impact of query fanout

In this subsection, we focus on testing the impact of the query fanout. We present two cases, i.e., a single class case and a two-class case. Consider a cluster of size N=100 and three different types of queries corresponding to three different fanouts 1, 10 and 100, similar to the testing scenario in [91], in which fanouts 1, 8 and 33 are used. Further assume $P(1)$=100/111, $P(10)$=10/111, and $P(100)$=1/111, i.e., the probability for a fanout

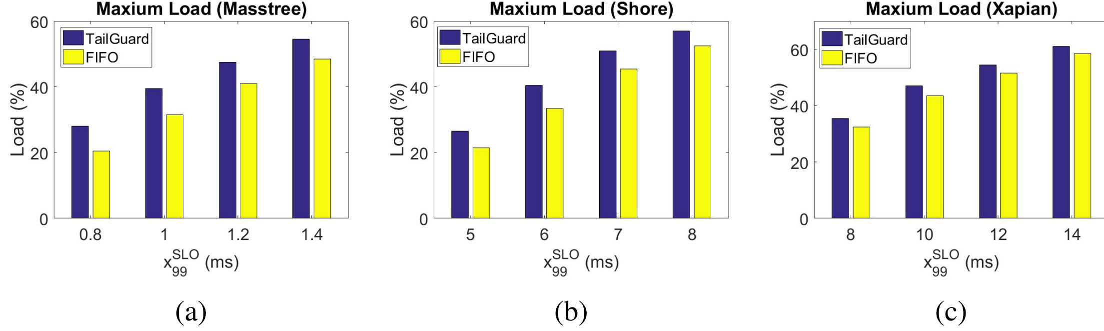


Figure 3.4: The maximum loads with a single service class.

is inversely proportional to the fanout itself, similar to the one observed by Facebook [89]. This makes the total numbers of tasks from the three query types to be, on average, the same. For a given tail latency SLO of x_{99}^{SLO}, the task pre-dequeuing time budget for a query with fanout k_f (1, 10 or 100) is T_b=$x_{99}^{SLO} - x_{99}^{u}(k_f)$.

Note that meeting the tail latency SLO for queries as a whole does not guarantee that queries of individual types can meet the tail latency SLO. Hence, in the following simulation, we measure the tail latency for each type of queries and identify the maximum load at which all three types of queries meet their tail latency SLOs.

We first consider the case with a single service class, i.e., all the queries have to meet a single SLO. In this case, both PRIQ and T-EDFQ behave exactly the same as FIFO and hence, we only compare TailGuard against FIFO. Figure 3.4 depicts the maximum loads that can meet the tail latency SLO for TailGuard and FIFO for four different tail latency SLOs (these SLOs are chosen such that the corresponding maximum loads fall in the range of 20% to 60% which are the typical system loads for the current commercial clouds serving DU applications [122, 48]). This gives us a good idea about TailGuard's performance gain/loss with respect to those of the currently practiced ones. As we can see, for all the cases, TailGuard achieves higher loads compared to FIFO, while meeting the same tail latency SLO. The performance gain increases as the tail latency SLO becomes tighter. This is because a query with a higher fanout has a tighter task queuing deadline and hence, higher chance to violate the tail latency SLO. Therefore, TailGuard that reorders the tasks based on queuing deadlines can help meet the tail latency SLO for all queries, resulting in higher performance than FIFO, especially when the tail latency SLO becomes more stringent. For example, for Masstree, the maximum load increases from 20% for FIFO to 28% for TailGuard at $x_{99}^{SLO} = 0.8ms$, resulting in about 40% higher resource utilization. In other words, TailGuard can save 40% task server resources over FIFO (also PRIQ and T-EDFQ), while meeting the same tail latency SLO, hence reducing the cost.

		K_f= 1	K_f= 10	K_f=100
x_{99}=0.8	FIFO	0.439	0.594	0.798
	TailGuard	0.572	0.745	0.797
x_{99}=1.0	FIFO	0.533	0.731	0.997
	TailGuard	0.705	0.941	0.994
x_{99}=1.2	FIFO	0.647	0.889	1.192
	TailGuard	0.817	1.098	1.193
x_{99}=1.4	FIFO	0.751	1.061	1.389
	TailGuard	0.945	1.262	1.392

Table 3.3: The 99th tail latency (ms) of three types of queries at maximum loads for the Masstree workload.

To gain more insights, for Masstree, Table 3.3 gives the breakdowns of the tail latencies at the maximum loads for the three types of queries. First, we note that at the maximum loads, the query type with k_f=100 barely meets the tail latency SLOs for both schemes. In other words, the maximum achievable load for both queuing policies are constrained by the query type with the highest k_f. For the other two query types, the tail latencies are smaller than the corresponding tail latency SLOs, implying that they get more resources than they need, especially for the one with k_f=1. The performance gain for TailGuard comes from more balanced resource allocation among the three types, as evidenced by the closer tail latencies among the three types than those for FIFO. The results clearly indicate that the query fanout has to be taken into consideration in task resource allocation for meeting query tail latency SLO to maximize the system performance.

Now we consider the case with two service classes with the tail latency SLO of the lower class being 1.5 times of that of the higher class, i.e., $1.5x_{99}$, where x_{99} is the tail latency SLO for the higher class. Each query is randomly assigned to one of the two classes with equal probability. Both the Poisson and Pareto arrival processes are considered. Due to limited space, only the results for the Masstree workload are given (the results for the other two workloads are similar).

Figure 3.5 shows the maximum loads under which all queries can meet their tail latency SLOs. From the results (Figure 3.5 (a)) with the Poisson arrival process, we can see that the performance gains of TailGuard over FIFO increase to up to 80%, much higher than that in the single class case (i.e., up to 40%). FIFO treats every task equally. Hence its performance is constrained by the most resource demanding queries, i.e., the higher class queries with the largest fanout. The TailGuard performance gain is up to 40% with respect to PRIQ. PRIQ gives higher priority to the higher class queries, resulting in lower class queries having less resources to meet their tail latency SLOs. The TailGuard performance

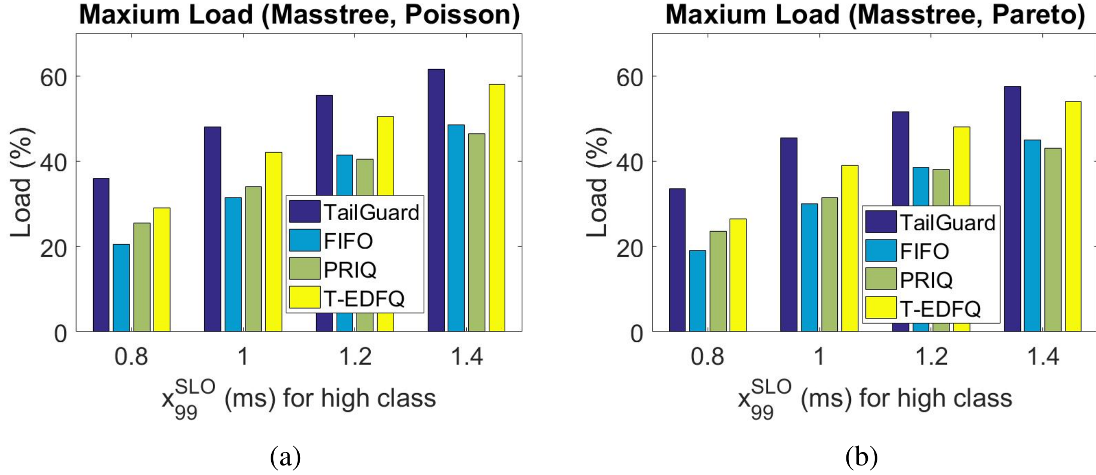


Figure 3.5: The Maximum loads with two classes for the Masstree workload: (a) Poisson and (b) Pareto arrival process.

gain is up to about 22% with respect to T-EDFQ, smaller than that with respect to PRIQ. This means that by incorporating the actual tail latency SLO, rather than just the class information, T-EDFQ can allocate task resources more accurately than PRIQ does. In turn, TailGuard improves over T-EDFQ by further incorporating query fanout information in task resource allocation.

The performance gains for TailGuard against the other three schemes with the Pareto arrival process (Figure 3.5(b)) are similar to those with the Poisson arrival process. Meanwhile, the maximum loads decease about 2% to 6% for all schemes compared to those with the Poisson arrival process. This means that the burstiness of query arrivals mainly impact the overall achievable load, but much less on the relative performance of different queuing policies. Hence, in the following cases studies, we only present those with the Poisson arrival process.

3.4.3 Impact of service class

Again, consider the cluster of size N=100. Now all queries have the same fanout of k_f=100, i.e., for each query, its tasks are served by all the task servers in the cluster in parallel, which is the case for OLDI services. We evaluate the performance of TailGuard for workloads with two different service classes, denoted as Class I and Class II. The tail latency SLOs for Class I/II are 1/1.5, 6/10 and 10/15 ms for Masstree, Shore and Xapian, respectively. Again, these tail latency SLOs are chosen such that the achievable maximum load ranges from 20% to 60%. A query has equal probability to request for either of the

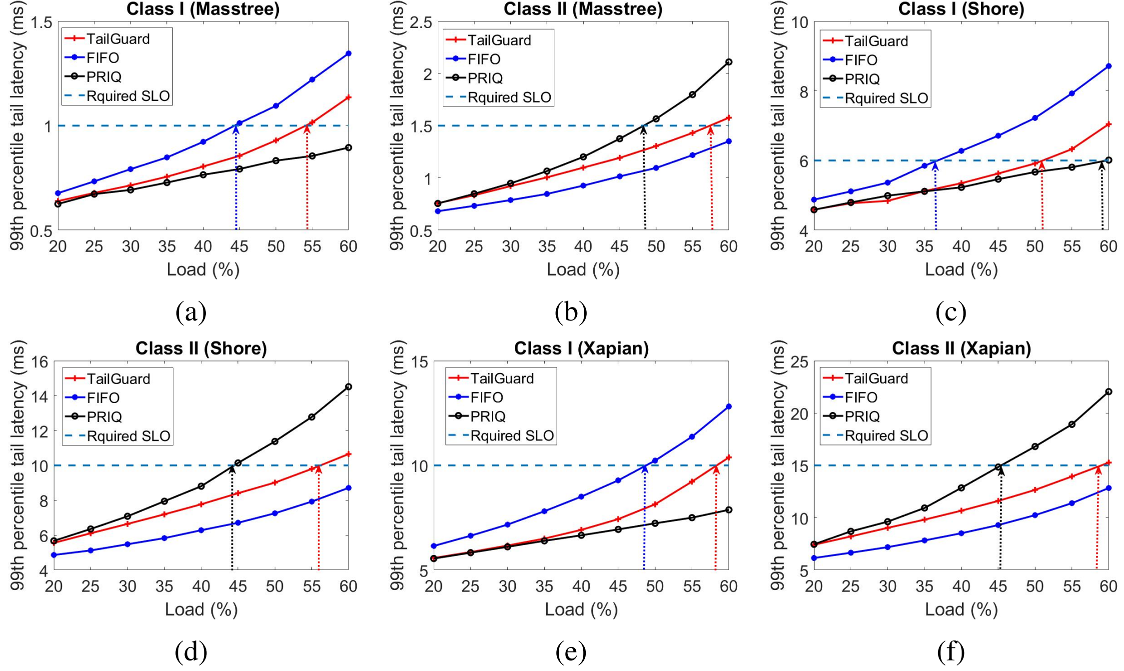


Figure 3.6: The 99th percentile latency at different loads. The cyan line indicates the required tail latency and the arrows points to the maximum load that the tail latency can be met.

two classes. For any query of a given class, by substituting the corresponding x_{99}^{SLO} and $x_{99}^{u}(100)$ from Table 3.2 into Eq. (3.7), we arrive at the task pre-dequeuing time budgets. For example, for Masstree, the budgets for classes I and II are 1-0.473=0.527ms and 1.5-0.473=1.027ms, respectively. As the fanout is the same for all queries, T-EDFQ behaves the same as TailGuard, and hence we compare the performance of TailGuard against both FIFO and PRIQ.

Figure 3.6 presents the simulation results. For each plot, the cyan dash line represents the tail latency SLO for that class and the arrows, each having the same color as the tail latency curve for a queuing policy, indicate the maximum achievable loads that meet the tail latency SLOs.

As one can see, for all three workloads, FIFO, which is class unaware, gives equal resources to queries from both classes. Since the task resource demands or task pre-dequeuing time budgets for tasks from classes I and II are quite different, e.g., 0.527ms and 1.027ms, respectively, as calculated above, for Masstree, indiscriminately allocating equal resources to tasks results in a very low achievable load for class I queries but very high achievable load for class II queries, e.g., 45% for class I, as shown in Figure 3.6(a), and higher than 60% for class II, as shown in Figure 3.6(b). Consequently, to meet the tail

latency SLOs for both classes, FIFO allows the cluster to run at 45% for Masstree, 36% for Shore (see Figure 3.6(c)) and 49% for Xapian (see Figure 3.6(e)).

PRIQ, on the other hand, is class aware, but it gives strict priority to tasks in Class I over Class II. This results in unbalanced resource allocation in favor of Class I over Class II. Consequently, the maximum load for class II is about 48% for Masstree, and about 45% for both Shore and Xapian, while the maximum load for class I reaches more than 60% for all three workloads. Again, the low load for class II limits the overall achievable load that meets both tail latency SLOs.

In contrast, as a class-aware approach and with task budgeting, TailGuard can balance the resources allocated to tasks closely in proportion to their resource demands, resulting in much closer maximum loads for the two classes (i.e., within 5% difference) for all three workloads. As shown in Figure 3.6, the maximum loads for Classes I and II for Masstree/Shore/Xapian are about 54%/51%/58% and 57%/56%/ 59%, respectively. Hence, the maximum loads that meet both tail latency SLOs are 54%/51%/58% for the three workloads, respectively. The performance gain of TailGuard over FIFO and PRIQ are up to 40% (i.e , from 36% to 51%) compared to FIFO and up to 30% (i.e., from 45% to 58%) compared to PRIQ.

3.4.4 Joint with Outlier Alleviation Solution

As we mentioned earlier, TailGuard is orthogonal and complementary to most existing outlier-alleviation solutions. To demonstrate this, we test the performance of TailGuard, along with the Adaptive Slow-to-Fast task scheduling scheme (called ASF in this paper) based on DVFS [52]. In ASF, a task server starts to serve a task at a low power level and switches to a higher power level to shorten the task execution time if a task service time runs longer than a threshold. The goal of ASF is to alleviate the impact of outliers, while minimizing the power consumption. In our simulation, a task runs at a normal (low) power level until its service time reaches twice the mean task service time, when it switches to run at a higher power level so that the remaining task service time is reduced by half (i.e., the task service speed doubles). Clearly, TailGuard is orthogonal to ASF. From TailGuard's point of view, the only difference ASF makes is that the CDF of the task service time, $F(t)$, seen by TailGuard is changed to a new one with a shorter tail. Hence ASF helps TailGuard to achieve either a higher query throughput or tighter query tail latency SLOs. We use the same case given in Section 4.3 to test it. Again, we only present the results for the Masstree workload due to space limitation.

Figure 3.7 depicts the performance of TailGuard compared with FIFO and PRIQ. We can see that all three schemes can run at higher loads to support the required tail latency

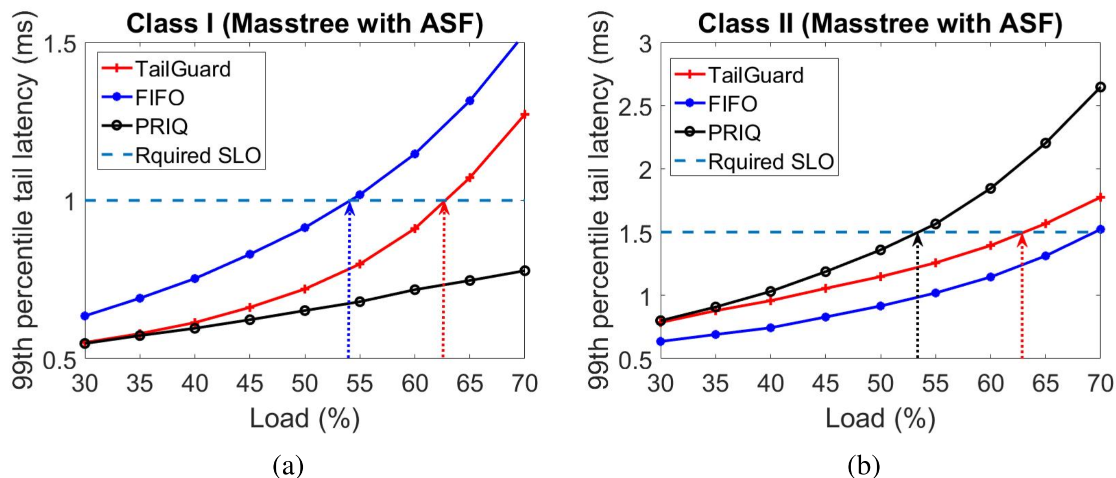


Figure 3.7: TailGuard with ASF.

SLOs, thanks to ASF (see Figure 3.6 (a) and (b)). The performance gains of TailGuard over FIFO and PRIQ are almost the same as that without ASF. Namely, the maximum load with guaranteed tail latency SLO increases from about 54%, 44% and 48% to about 63%, 54% and 54% for TailGuard, FIFO and PRIQ, respectively. These results demonstrate that TailGuard may indeed work with some of the outlier-alleviation solutions seamlessly to further improve the performance.

3.4.5 TailGuard with Query Admission Control

Now we test the TailGuard query admission control scheme. Consider the same case presented in Section 4.3 (only the result of Masstree is given due to limited space). We first run TailGuard without admission control to find the task queuing deadline violation threshold R_{th} at the maximum acceptable load when TailGuard can barely provide the tail latency SLO guarantee. The maximum acceptable load thus found is about 54% and the corresponding threshold is 1.7%. We use a moving window with size of 1000 queries (or 100000 tasks) to compute the task queuing deadline violation ratio.

Figure 3.8 shows the accepted/rejected loads and the query tail latencies at different loads. First, we see that the query tail latency SLOs for both classes are guaranteed at all loads. When the load is over the maximum acceptable loads, the query tail latency of Class I closely approaches its tail latency SLO, while the tail latency of Class II is a little below its SLO. This is due to the fact that Class I tasks have tighter pre-dequeuing time budgets to meet and hence have higher chances to miss the queuing deadlines as we explained in Section 4.3. Second, we note that the accepted loads (the load is computed using the

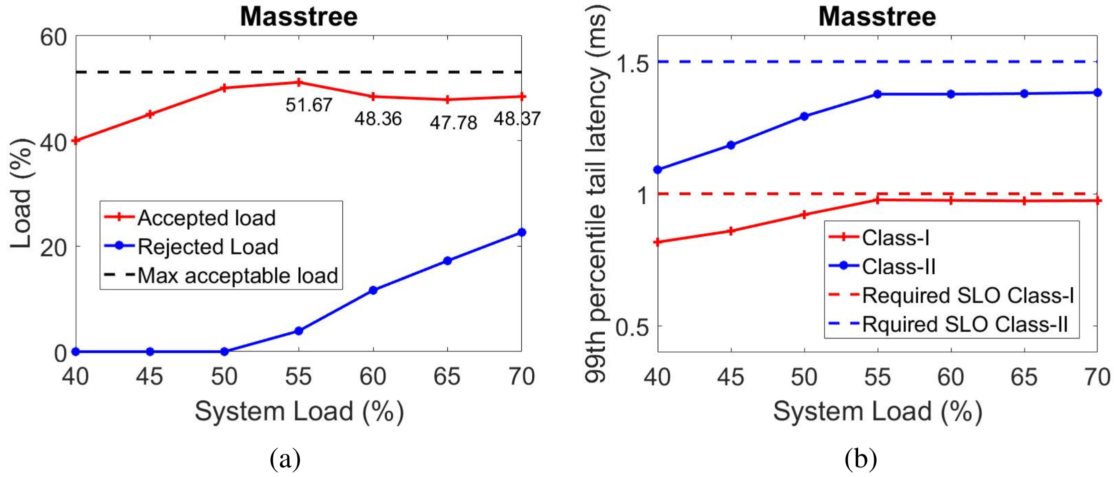


Figure 3.8: TailGuard with query admission control. (a) is the accepted/rejected load; and (b) is the query tail latency for Class I and II.

accepted queries only) closely approach its respective maximum acceptable loads (within 2.5%). Further increasing the load beyond the maximum acceptable loads, the accepted load drops to about 6% below the maximum acceptable loads. There are two reasons for this to happen. First, TailGuard may not drop the exact number of queries needed to perfectly meet the tail latency SLO. Second, just like any feedback loop control solutions, TailGuard incurs a delay between the measurement and control, which inevitably makes the achievable load to be lower than the maximum acceptable load. Nevertheless, these results demonstrated that the TailGuard query admission control can indeed provide tail latency SLO guarantee, while maintaining high resource utilization.

Finally, we note that the simulation results for cluster size, N=1,000, and in the presence of 4 classes are also available and consistent with the ones above, which however, are not presented here for the lack of space.

3.4.6 Evaluation in the Amazon EC2 cloud

To verify the simulation results, with all possible delays (e.g., dispathcing and communication delays) accounted for, we implement and test TailGuard in a small cluster in the Amazon EC2 cloud. The TailGuard implementation includes the query scheduling code and the Spark plug-in code. TailGuard is implemented by modifying the open source implementation codes for Eagle [34] and Pigeon [131]. The query scheduler is deployed at the application front-end, exposing services to allow the framework to submit query scheduling requests using remote procedure calls (RPCs). Apache Thrift [120] is used by all RPCs

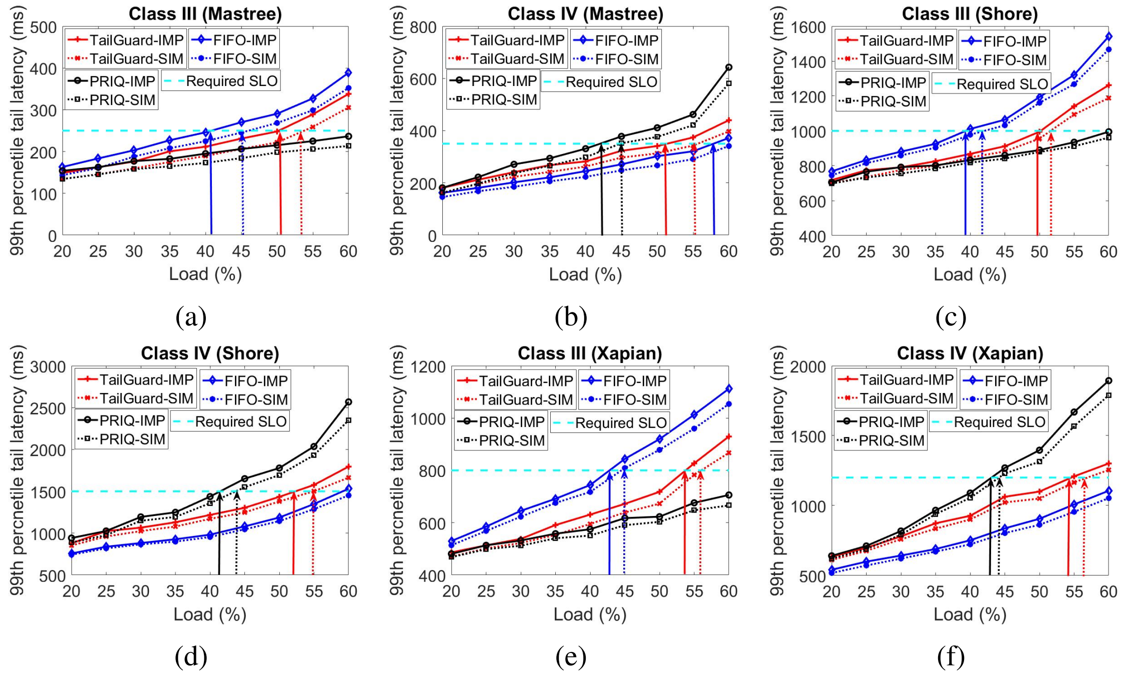


Figure 3.9: Amazon EC2 implementation (IMP) vs simulation (SIM).

for internal communications between modules of a scheduler. Both the query handler and task servers are all hosted on m4.large instances.

The cluster is composed of 1 query handler and 100 task servers. The task non-service time t_x that accounts for the query processing delay, task dispatching delay, task result returning delay, and merging processing delay turns out to be much larger than the mean service times for all three workloads, which are in sub-milliseconds. Hence, to alleviate the domination of the task response time by t_x, for our experiment, we increase the mean task service times by 300, 200 and 100 times, i.e., 52.8, 68.3 and 92.5ms for Mastree, Shore and Xapian, respectively. With this setup, the task service time is on the same order as the task non-service time. Consider two service classes with the corresponding tail latency SLOs 250/350, 1000/1500 and 800/1200 ms for Class III/IV in Mastree, Shore and Xapian workloads, respectively. Similar to Case 4.3, all queries have the same fanout 100.

Figure 3.9 presents the results for both experiments, denoted as IMP, and simulation, denoted as SIM. As expected, TailGuard achieves better performance than both FIFO and PRIQ for all the three workloads. Clearly, the experiment results are consistent with the simulation results, with the differences within 15%. The difference increases as the load increases. This is because t_x, which is overlooked in simulation, becomes larger for the experiment as the load increases. The increased t_x value makes the tail latency for the

experiment alway larger than that for the simulation. Nevertheless, we can see that the performance gains of TailGuard over FIFO and PRIQ for the experiment are almost the same as those for the simulation for all the three workloads. These results, to some extent, verify the effectiveness and accuracy of simulation results presented in the previous sections.

3.4.7 Evaluation in an SaS Testbed

Finally, we evaluate and compare TailGuard against the other three schemes in an on-campus SaS testbed being developed.

Testbed Setup: The testbed is currently composed of four clusters of edge nodes, located in four rooms in two buildings, including a server room and a Graduate Research Assistant (GRA) office next to a wet lab in one building, and a faculty office and a Graduate Teaching Assistant (GTA) office in another building. Each of these four clusters, referred to as Server-room, Wet-lab, Faculty and GTA clusters hereafter, consists of 8 Raspberry Pi devices, serving as edge nodes, with each currently attached with a temperature sensor and humidity sensor and connected to the Internet through an Ethernet switch. Each edge node receives sensing data from both sensors periodically and keeps up to eighteen-month-worth of the data records. Since the Wet-lab cluster may require low delay sensing data, we use the higher performing Raspberry Pi's to furnish the cluster than the ones for the other three and have the query handler co-located with the cluster to minimize the communication delay.

Use Cases: We consider three likely use cases belonging to three distinct classes, A, B, and C, to stress test TailGuard, with the 99th percentile tail latency SLOs equal to 800, 1300, and 1800 ms, respectively.

First, we note that the server room and wet lab are shared by many research groups and individuals, who may want to closely monitor individual devices they own to track the sensing data. This use case can stress test TailGuard by generating heavier workload on these two clusters than the other two. To create even more unbalanced load, instead of evenly distributing the load on these two clusters[6], we place 80% of such workload on the Server-room cluster and the rest 20% randomly assigned to the others. Moreover, queries of this use case are considered class A with the most stringent tail latency SLO and constitute 50% of the total queries.

Second, we consider a use case targeting at potential users who may want to get an overall reading of the temperature and humidity in all areas with low delay. For such use case, a query fans out 4 tasks, each accessing a randomly selected edge node in a

[6]Note that equipped with the highest performing nodes and closest to the query handler, the Wet-lab cluster can hardly pose a performance bottleneck.

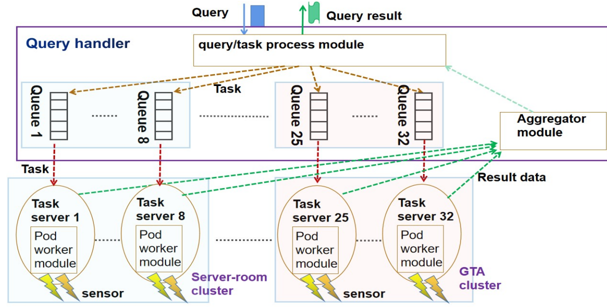


Figure 3.10: SaS testbed architecture.

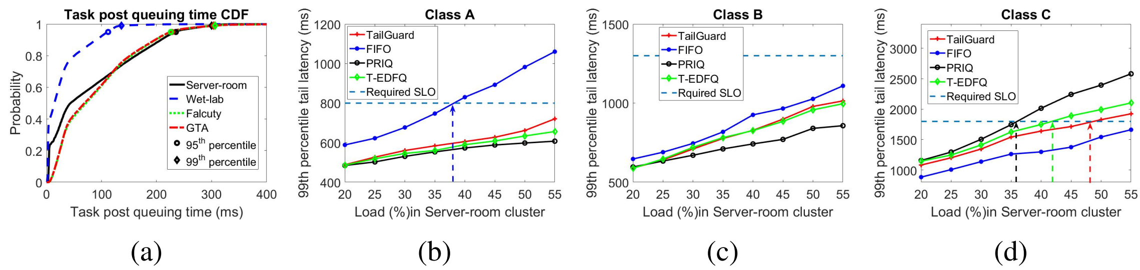


Figure 3.11: (a) The task post-queuing time CDFs in four clusters. Circle and diamond represent the 95th and 99th percentile tail latencies, respectively. (b), (c) and (d) are the 99th percentile query tail latency of the three classes at various loads.

separate cluster. This use case is considered less time critical than the previous one and thus designated class B. We assume that it takes up 40% of the total queries.

Third, some users may require detailed, relatively longer term sensing data records to be retrieved from all edge nodes with a loose tail latency SLO. Hence, all the queries in this use case have fanout 32 and are assigned as class C, and 10% of the total queries are assigned to this class.

SaS testbed Architecture: Figure 3.10 depicts the SaS testbed architecture. The query handler runs in a PC and consists of a query/task process module and an aggregator module. Queuing takes place centrally in the query/task process module with 32 sets of queuing buffers allocated, one for each edge node. The testbed resources are managed by K3s [1], which orchestrates the pod resource allocation in edge nodes. All the communications between the query handler and an edge node use keep-alive HTTP/1.1 connections.

A task arriving at an edge node retrieves one or multiple temperature and/or humidity records from the local database. It has an equal probability of retrieving one to up to thirty-day-worth of consecutive records starting from a random time in the eighteen-month period. After retrieving the records, the edge node sends the records to the aggregator module as well as an edge-node-idle message to the process module. Upon receiving the

records for all the query tasks, the aggregator merges the records for the query, which are finally sent to the user.

To further test if TailGuard can perform well with inaccurate CDFs of unloaded task post-queuing times, we let all 8 edge nodes in each cluster share the same CDF based on the samples evenly collected from all edge nodes in the cluster. Figure 3.11 (a) presents the CDFs for all four clusters. First, we note that the CDFs (red and green lines) for Faculty and GTA clusters are almost identical, as they use the same model of Raspberry Pi's and located in the same building. With the same model of Raspberry Pi's but located in a different building and closer to the query handler, the CDF for the Sever-room cluster concentrates more in the lower post-queuing time region than the previous two. In contrast, equipped with the highest performing Raspberry Pi's and co-located with the query handler, the Wet-lab cluster offers significantly smaller overall post-queuing time than the other three. More specifically, the mean, 95th and 99th task post-queuing times are about 82/31/92/91, 235/112/226/228, and 300/136/306/304 ms for the Server-room/Wet-lab/Faculty/GTA clusters, respectively, making the system highly heterogeneous.

Results and Analysis: Figure: 3.11 (b), (c) and (d) present the results. We note that TailGuard, FIFO, PRIQ and T-EDFQ can achieve the maximum load of about 48%, 38%, 36% and 42%, respectively. This results in the performance gains of TailGuard over FIFO, PRIQ and T-EDFQ to be 26.3%, 33.3% and 14.3%, respectively. As one can see, both the performance gains and the maximum load differences in such a highly heterogeneous system are in line with the simulated and the Amazon EC2 ones (homogeneous systems).

The above stress test, together with the simulation and Amazon EC2 test, demonstrates that TailGuard is effective to improve resource allocation performance for DU applications, even in a heterogeneous system with highly unbalanced workload patterns, and varied processing and communication delays.

All source codes will be made available online after the paper is published.

3.5 Conclusions

In this paper, we propose TailGuard for data-intensive user-facing applications, aiming at maximizing resource utilization, while providing tail latency SLO guarantee. TailGuard decouples the upper query level design from the lower task level design. First, at the query level, a decomposition technique is developed to compute the task queuing deadline for a query with the given tail latency SLO and fanout. Second, at the task level, based on the task queuing deadline, a simple earliest-deadline-first queuing policy is employed to manage task queues to improve the resource utilization. TailGuard is evaluated by simulation using three Tailbench workloads as input. The results demonstrate that TailGuard can

improve resource utilization by up to 80% while meeting tail latency SLOs, compared to the FIFO, PRIQ and T-EDFQ queuing policies. TailGuard is also implemented and tested in the Amazon EC2 cloud and a heterogeneous SaS testbed and the test results agree with the simulated ones.

CHAPTER 4

Performance Models for Data-intensive, User-facing Workloads with Query Tail Latency SLO

Data-intensive, User-facing Services (DUSes), such as web searching and online social networking, must meet stringent query tail-latency service level objectives (SLOs) and deal with scale-out workloads with each query spawning potentially a large number of tasks (i.e., query fanout) to be processed in parallel. Meanwhile, a DUS service provider needs to maximize its revenue or net profit, by minimizing its resource allocation. However, achieving this dual design objective is challenging, as there is no existing mathematical underpinning that can capture the resource demand for such workloads, especially in the presence of a wide range of possible cluster design and configuration options in practice. In this paper, we develop a unified mathematical model that provide a direct link between query tail latency, throughput and resource demand under a range of system design and configuration options, including vertically scaled versus horizontally scaled clusters, with or without tail cutting, and scale-out versus scale-up workers in the cluster, with or without redundant task issues. With this model, we are able to derive the maximum sustainable cluster loads at different query tail latency for different design and configuration options. We also find that under scaling conditions, there is a cross-over load beyond which the horizontal scaling/scale-out outperforms vertical scaling/scale-up at the cluster/worker level. The accuracy of the proposed model in predicting the DUS performance is verified by extensive simulation, making it a useful mathematical tool to facilitate the effective DUS resource planning in cloud.

4.1 Introduction

Data-intensive, User-facing Services (DUSes), such as web searching, digital marketing, online social networking, and online retailing, have emerged as predominate workloads in clouds and datacenters. Meeting stringent query tail-latency Service Level Objectives (SLO) for queries of a DUS has been widely recognized as a critical requirement, which has an enormous impact on user experience, application adoption, customer satisfaction, and ultimately, business revenue and net profit. For example, at Shopzilla, reducing query tail latency from seven seconds to two seconds increases page views by 25%, and

revenue by 7%. And, for Amazon online web services, every 100-millisecond addition of query tail latency causes 1% decrease in sales [7]. As a result, from a DUS service provider's point of view (e.g., a DUS service provider who wants to deploy a DUS by renting virtual machine instances in Amazon EC2 cloud), an important design objective is to allocate the minimum amount of the cluster resources to satisfy a desired given query tail latency SLO and a target query throughput.

However, it is challenging to achieve the above design objective for two major reasons. First, DUSes are driven by queries that require query responsiveness in sub-seconds to seconds and they have to deal with scale-out workloads, i.e., a query may need to touch on massive datasets that are partitioned into shards and distributed to a large number of workers (i.e., servers) in the cluster. In other words, a query may spawn a number of tasks (known as query fanout) to be sent to some or all of the workers in the cluster to be queued and processed in parallel. The query response time is determined by the slowest task of the query. The difficulty lies in the fact that to meet a given query tail-latency SLO, the resource demands for queries with different fanouts are different. For example, assume that with a given amount of resource allocated to process each task, there is 1% probability that the task response time will be over $100ms$. Then, at query fanout, k_f, the probability that the response time for the query will be over $100ms$ is $1 - 0.99^{k_f}$. This means that a query with $k_f = 1$ and $k_f = 100$ have 1% and 63.4% probabilities of being over 100 ms. This example implies that with the given per-task resource allocation, a query with $k_f = 1$ can exact meet the tail latency SLO in terms of the 99th-percentile query latency of 100 ms, whereas a query with $k_f = 100$ cannot. In order to meet the same tail-latency SLO, its tasks must be allocated a much larger amount of resources. This clearly demonstrates that to achieve the above design objective, the cluster resource allocation must take query fanouts into account. Unfortunately, none of the existing datacenter workload models attempts to provide a link between the query fanout and task resource demand (see Section 4.4 for more details).

Second, there are a wide range of possible cluster design and configuration options and different options may lead to vastly different query tail latency and throughput performance. We identify some widely adopted design and configuration options, as given in Table 4.1, in terms of, scale-up or scale-out, with or without redundant task issues [126] at the worker level; and vertical scaling or horizontal scaling with or without tail cutting [61] at the cluster level. This leads to a total of 16 distinct design and configuration options. It becomes apparent that it is impractical to design and configure a cluster to exhaust all the options and then identify which one should be adopted to achieve the above design objective. Instead, one must resort to model-based approaches that can help quickly compare different possible options to identify the most promising few before field trial. Al-

Table 4.1: System Configuration

System Configuration	Worker Level	Cluster Level
Scaling mode	Scale-up Scale-out	Vertical scaling Horizontal scaling
Tail optimize policy	Redundant issue	Tail Cutting

though there have been some recent efforts made along this line based on queuing models [130, 104], they have limited scope, focusing on a few design and configuration options, e.g., scale-out versus scale-up, or with or without redundant task issues (see Section 4.4 for more details).

In this paper, we develop a unified queuing network model that provides a direct link between cluster resource demand, query tail-latency SLO and throughput for all design and configuration options listed in Table 1 and scale-out workloads that characterize DUSes. Other main contributions main of this paper include:

1. We derive the maximum sustainable cluster loads at different query tail-latency-to-mean ratios for different design and configuration options;
2. We prove that under certain resource scaling conditions, there is a worker-level cross-over load, below (above) which the scale-up (scale-out) workers outperform scale-out (scale-up) ones, independent of query fanout;
3. We prove that under certain resource scaling conditions, there is a cluster-level cross-over load, a function of query fanout, below (above) which the vertical-scaling (horizontal-scaling) outperforms horizontal-scaling (vertical-scaling);
4. We perform comprehensive test of the accuracy of the proposed model in predicting the DUS performance by simulation.

We also briefly discuss how the proposed model may potentially be used to aid highly effective cluster resource planning for DUS service providers who rent cloud resources to enable DUSes.

The rest of the paper is organized as follows. Section 4.2 describes the models and the main results. Section 4.3 verifies the accuracy of the proposed model by extensive simulation. Section 4.4 reviews the related work. Finally, Section 4.5 concludes the paper.

4.2 A Unified Model

In this section, we first introduce the cluster and worker level model components in Section 4.2.1. Then we describe the performance scaling model components that establishes a link between resource scaling and performance for different system design and

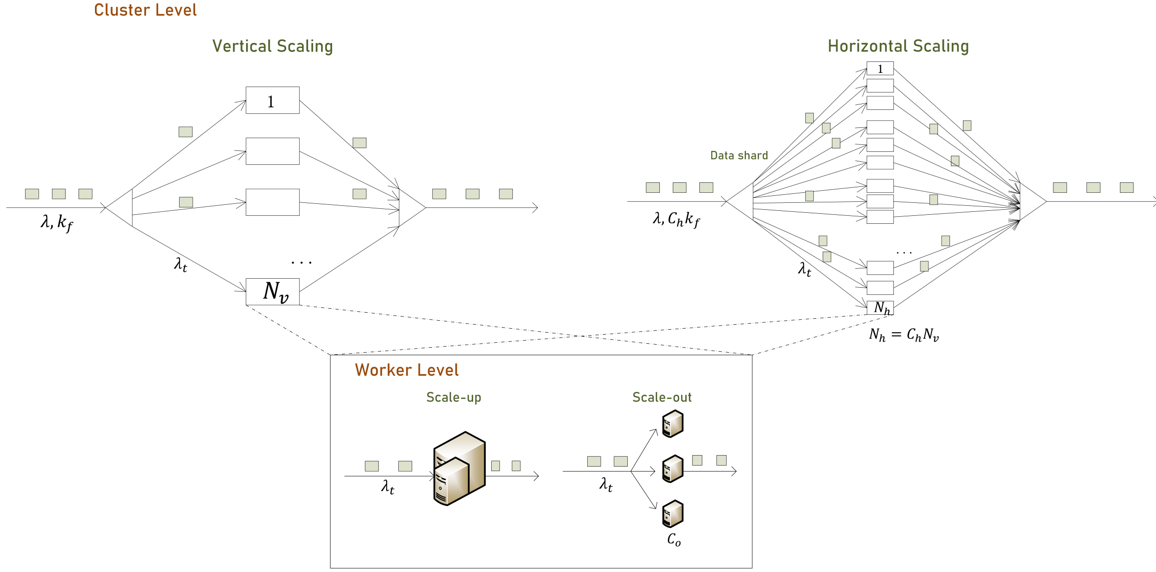


Figure 4.1: Fork-Join Structure, cluster level, and worker level scaling model

configuration options in Section 4.2.2. Based on the unified model given in these two subsections, a special case that can capture the performance bounds is derived in Section 4.2.3. As the unified model generally works for any task response time distributions, a solution as to how to estimate the task response time distribution with minimum measurement cost in practice is given in Section 4.2.4. The symbols used in this paper are listed and defined in Table 4.2.

4.2.1 Model for Cluster Scaling

First, consider the cluster level scaling. The upper left plot in Figure 4.1 depicts a fork-join queuing network model for a vertically scaled cluster with a total number of N_v fork nodes representing N_v high-performance workers, each hosting a big data shard of the total dataset. The query arrivals follow a random arrival process. Each query spawns k_f^v tasks on average, mapped to k_f^v out of N_v fork nodes ($k_f^v \leq N_v$) with equal probability (to limit the exposure, in this paper, we only consider homogeneous workload), resulting in an average task arrival rate, $\lambda_t = \frac{k_f^v}{N_v}\lambda$, at each fork node. The average task workload is denoted as $E(W)$.

The horizontally scaled cluster model, depicted in the upper right plot in Fig. 4.1, replaces each fork node in the vertically scaled cluster model with C_h fork nodes, representing C_h low-performance workers, resulting in a total number of $N_h = C_h N_v$ workers in the cluster. It models the case where each worker hosts a data shard equal to one-C_hth

Table 4.2: Symbol and Description

Symbol	Description	Symbol	Description
λ	Average query arrival rate	ρ	Worker Load
λ_t	Average task arrival rate	ρ_{out}	The load of Scale-out worker
N_h	Total number of fork nodes in vertically scaled cluster	ρ_{up}	The load of Scale-up worker
		ρ_c	Scale-up-vs-scale-out crossover load
N_h	Total number of fork nodes in horizontally scaled cluster	ρ'_c	Vertical-vs-horizontal crossover load
		c	Number of processing unit in M/G/c queue
C_o	Number of processing units in scale-out worker	ρ_h	Load for horizontally scaled cluster
C_h	Number of horizontally scaled workers	ρ_v	Load for vertically scaled cluster
$E(S)$	Mean task service time	x_p	pth-percentile query tail latency
$E(S_h)$	Mean task service time for horizontally scaled cluster	x_p^{out}	pth-percentile tail latency of scale-out worker
$E(S_v)$	Mean service time for vertically scaled cluster	x_p^{up}	pth-percentile tail latency of scale-up worker
μ_u	Average task service rate for scale-up worker	x_p^h	pth-percentile query tail latency for horizontally scaled cluster
μ_o	Average task service rate for scale-out worker	x_p^v	pth-percentile query tail latency for vertically scaled cluster
$E(W)$	Average task workload	n	Number of task issues for redundant-task-issue
$E(W_v)$	Average task workload for vertically scaled cluster	p	pth-percentile
$E(W_h)$	Average task workload for horizontally scaled cluster	γ_{tm}	Query tail-to-mean ratio: $\frac{Tail\ latency}{Mean\ service\ time}$
v	Workload processing speed at a processing unit	k_f	Query fanout
v_h	Workload processing speed at a processing unit in horizontally scaled cluster	k_f^h	Query fanout in horizontally scaled cluster
		k_f^v	Query fanout in vertically scaled cluster
v_v	Workload processing speed at a processing unit in vertically scaled cluster	α	Shape parameter of the generalized exponential distribution
δ	Positive scaling factor at worker level	β	Scale parameter of generalized exponential distribution
ζ	Positive scaling factor at cluster level		

of the big data shard in the vertically scaled cluster. Accordingly, each query fans out to $k_f^h = C_h k_f^v$ fork nodes on average to reflect the fact that a query sent to either a horizontally or a vertically scaled cluster should touch upon the same portion of the dataset. As a result, the average task arrival rate, $\lambda_t = \frac{C_h k_f^v}{C_h N_v}\lambda = \frac{k_f^v}{N_v}\lambda$, the same as that in the vertically scaled cluster and the average task workload is now $E(W)/C_h$, assuming that the workload is linearly proportional to the shard size (e.g., for a search engine, the number of indices to be searched reduces by half, as the index shard size is cut by half).

Second, let's look at the worker level scaling, depicted in the lower middle plot in Fig. 4.1. Each fork node is either modeled as a G/G/1 queuing server or G/G/c queuing server. The former models a scale-up worker with a single task queue and a single high-performance processing unit that processes tasks one at a time. And the latter models a scale-out worker with a single task queue and C_o low-performance processing units that can process up to C_o tasks in parallel. For both server models, the first and second G represent any given random arrival process and task service time distribution, respectively.

A Unified Model: Now, denote the cumulative distribution function (CDF) of the task response time for tasks at any fork node generally as $F_T(t)$ for both scale-up (G/G/1) and scale-out (G/G/c) workers (We only consider homogeneous workers in the cluster). Also denote the average query fanout and cluster size for both vertically scaled and horizontally scaled clusters generally as, k_f and N, which equal k_f^v and N_v for vertically scaled

cluster and k_f^h and N_h for horizontally scaled cluster. Then we have the following generally results,

Lemma 1: The query response time distribution $G(t)$ can be approximately written as,

$$G(t) \approx F_T(t)^{k_f} \tag{4.1}$$

Proof: As the slowest task of the query determines the query response time, according to the order statistics[3], we have,

$$G(t) = Prob(MAX\{T_1, ..., T_{k_f}\}) = F_T(t)^{k_f}, \tag{4.2}$$

given that task response times T_i's for all k_f tasks are i.i.d. (i.e., independent and identically distributed) random variables. Although the i.i.d. assumption does not hold true for the tasks in a query, this approximation is found to be fairly accurate in capturing the query tail latency, as verified in Section 4.4. □

Lemma 2: The pth-percentile query tail latency of x_p is given by,

$$x_p = G^{-1}(\frac{p}{100}) \tag{4.3}$$

Proof: By definition, we have $\frac{p}{100} = G(x_p)$ and hence, Eq. 5.2, by taking the inverse function of $G(t)$. □

Tail latency with tail cutting: In this paper, tail cutting refers to a category of techniques that deal with slow tasks or stragglers [61]. Specifically, it refers to techniques that return partial results without waiting for the results from some slowest tasks, usually by setting a result return deadline, t_D, i.e., only the results returned by t_D will be included in the response to the query. We have the following result:

Lemma 3: With deadline t_D for tail cutting and task response time distribution, $F_T(t)$, the query response time distribution $G(t)$ is given by

$$G(t) \approx F_{T_{cutail}}(t)^{k_f} \tag{4.4}$$

where,

$$F_{T_{cutail}}(t) \begin{cases} \frac{F_T(t)}{F_T(t_D)} & \text{if} \quad t \leq t_D \\ 0 & \text{if} \quad t > t_D \end{cases} \tag{4.5}$$

Proof: With deadline t_D, the task response time distribution, $F_{T_{cutail}}(t)$, for a task is the task response time distribution, $F_T(t)$, truncated at $t = t_D$, and hence, is given by Eq. 4.5. For all the tasks of a query including those past the deadline t_D, they can be viewed as being able to finish by the deadline and hence, we have Eq. 4.4. Note that Lemma 2 still holds true in the case with tail cutting □

Tail latency with redundant task issues: Redundant task issues is another cat-

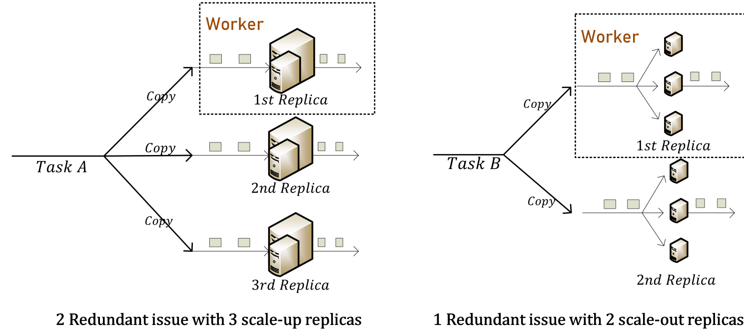


Figure 4.2: Worker level redundant issue policy

egory of techniques to combat slow tasks or stragglers [32, 129]. It takes advantage of the fact that for fault tolerance, in practice, a shard in a worker is copied to a few worker replicas, e.g., three for a scale-up worker and two for a scale-out worker in Figure 4.2. Specifically, the same task may be issued to one or more worker replicas if the task issued to the original worker does not finish by a certain deadline [32]. Then whichever task finishes first will result in the removal of the other tasks from other worker replicas.

In this paper, we consider an ideal algorithm that leads to the query tail performance upper bound for this category of techniques. Specifically, for each task arrival, the worker issues n copies of the task to be queued at n worker replicas, simultaneously (e.g., $n = 3$ and 2 for the scale-up and scale-out workers in Figure 4.2). The completion of the first task among all n copies of the task causes all the other copies of the task to be removed from the other worker replicas with no impact on their queuing performance. This is an ideal algorithm that cannot be realized in practice, simply because some of those being removed may be already in service and hence, have consumed resources of the corresponding worker replicas. As a result, this ideal algorithm provides a performance upper bound for this category of techniques. We have the following result,

Lemma 4: The query response time distribution with n task issues is,

$$G(t) = [F_{T_{min}}(t)]^{k_f}, \tag{4.6}$$

where $F_{T_{min}}(t) = 1 - [1 - F_T(t)]^n$ without tail cutting. With tail cutting, $F_T(t)$ should be replaced with $F_{T_{cutail}}$.

Proof: According to the order statistics, the task response time, $F_{T_{min}}(t)$, is:

$$F_{T_{min}}(t) = Prob(MIN\{T_1, ..., T_n\}) = 1 - [1 - F_T(t)]^n \tag{4.7}$$

By substituting $F_T(t)$ with $F_{T_{min}}(t)$ in Eq. 5.1, we arrive at Eqs. 4.6. □

4.2.2 Models for Resource Scaling

Scale-up versus scale-out Scaling: We assume that the resource allocated to a worker only impacts the average task service time, not the distribution function of the task service time. Let μ_o and μ_u represent the average task service rates (i.e., the inverse of the average task service times) for scale-out and scale-up workers, respectively. To be comparable, we assume that some cost measure, e.g., the capital expenditure per worker, for both scale-up and scale-out workers are the same, denoted as CX. Then we use the following performance scaling model to capture the relative performance of the two types of scaled workers,

$$\mu_u = C_o^{\delta}\mu_o, \tag{4.8}$$

where δ is a scaling factor and $\delta > 0$, meaning that the processing unit for the scale-up worker must be more powerful and hence, process tasks faster than the individual ones for the scale-out worker. For example, in the case of hardware scaling, i.e., scaling up or out by using brawny cores or wimpy cores of the same die size [79], we have $\delta = 0.5$ according to the Pollack's rule [6]. In general, δ can be obtained by measurement, e.g., in the case of software scaling using virtual machines (VMs), containers, or pods[105]. With this scaling model, we have,

Lemma 5:

$$\rho_{up} = C_o^{(1-\delta)}\rho_{out} \tag{4.9}$$

where ρ_{up} and ρ_{out} are the loads for scale-up and scale-out workers, respectively.

Proof: For the scale-out worker, since there are C_o processing units, the average arrival rate at each processing unit is λ_t/C_o. Then by definition, we have,

$$\rho_{out} = \frac{\lambda_t}{\mu_o C_o} \tag{4.10}$$

and for the scale-up worker, by definition, we have,

$$\rho_{up} = \frac{\lambda_t}{\mu_u}. \tag{4.11}$$

By applying the scaling model (i.e., Eq. 4.8) to these definitions, we can easily arrive at Eq. 4.9. □

Now, we have the following important result,

Theorem I: For $\delta > 1$, the scale-up worker outperforms the scale-out worker in terms of the tail-latency performance, in the entire load range of $\rho_{up} \in [0, 1]$. and for $\delta < 1$,There is a cross-over load, $\rho_c < 1$, independent of query fanout k_f, such that,

$$\begin{cases} x_p^{out}(\mu_o, \rho_{out}, k_f) = x_p^{up}(\mu_u, \rho_{up}, k_f), \ \rho_{up} = C_o^{(1-\delta)}\rho_{out} = \rho_c \\ x_p^{out}(\mu_o, \rho_{out}, k_f) > x_p^{up}(\mu_u, \rho_{up}, k_f), \ \rho_{up} = C_o^{(1-\delta)}\rho_{out} < \rho_c \\ x_p^{out}(\mu_o, \rho_{out}, k_f) < x_p^{up}(\mu_u, \rho_{up}, k_f), \ \rho_{up} = C_o^{(1-\delta)}\rho_{out} > \rho_c \end{cases} \tag{4.12}$$

where $x_p^{out}(\mu_o, \rho_{out}, k_f)$ and $x_p^{up}(\mu_u, \rho_{up}, k_f)$ are the pth-percentile query tail latencies for the scale-out and scale-up clusters, respectively.

Proof: First, consider a single worker with extremely low load, i.e., λ_t and hence, ρ_{up} and ρ_{out}, are extremely small. In this case, a task arriving at either a scale-up or scale-out worker has a high probability to be serviced immediately without queuing delay. Since $\delta > 0$, $\mu_u > \mu_o$, according to Eq. 4.8, with the same service time distribution, except a larger service rate, or equivalently, a smaller mean service time, it is clear that at any percentile, p_t, the task tail-latency for the scale-up worker must be smaller than that for the scale-out worker.

Now, for $\delta < 1$, since $\rho_{up} > \rho_{out}$, according to Eq. 4.9, as λ_t increases, ρ_{up} will approach 1 sooner than ρ_{out}. Namely, the queue length and hence, the task tail latency of the scale-up worker will approach infinity, as ρ_{up} approaches one, whereas the queue length and hence, the task tail latency of the scale-out worker are still finite. This proves that there must be a crossover load, $\rho_c < 1$, beyond which (i.e., ρ_{up}>ρ_c), the scale-out worker outperforms the scale-up worker.

For $\delta > 1$, since ρ_{up}<ρ_{out}, according to Eq. 4.9, the crossover cannot occur as the load increases, as the task tail latency of the scale-out worker increases faster than that of the scale-out worker.

Now, consider the query with fanout k_f. According to Eq. 4.4, if the tail for the task CDF, $F_{T_{eff}}(t)$, becomes heavier, so does the tail for the query, $G(t)$, regardless of the query fanout, k_f. Therefore, the crossover load, ρ_c, applies to queries as well, independent of query fanout, k_f. □

The above result can be used to make informed decision, with a single measurable parameter, δ, as to which types of workers should be adopted, scale-out or scale-up ones, at any given query tail-latency SLO and target throughput or equivalently, load. Moreover, the independence of query fanout of the above result makes it possible to decouple the scaling decision making at the worker level (i.e., scaling out or scaling up) from that at the cluster level (i.e., horizontal or vertical scaling), greatly reducing the number of design and configuration options to be compared.

Vertical versus Horizontal scaling Now we assume that CX is the cost per vertically scaled worker in a vertically scaled cluster with N_v workers. we further assume that in a horizontally scaled cluster with $N_h = C_h N_v$ workers, the cost per C_h workers is CX. This ensures that the total costs for both clusters are the same. Therefore, the relative performances of the two are comparable. We also assume that the average task workload, $E(W)$, linearly increases with the shard size. With this assumption and letting $E(W)$ be the average task workload for a vertically scaled worker, then $E(W)/C_h$ is the task workload for a horizontally scaled worker, because the shard size at a horizontally scaled worker is one-C_hth of that at a vertically scaled worker.

Furthermore, since the scaling decision making at the worker level can be decouple from that at the cluster level, the vertically and horizontally scaled cluster configurations to be compared should use the same types of workers, either scale-up (G/G/1) or scale-out (G/G/c) workers, and the workers in both clusters have the same number of processing units. Now we adopt the following scaling model:

$$v_v = C_h^{\zeta} v_h \tag{4.13}$$

where v_v and v_h are the task workload processing speeds at a processing unit in vertically and horizontally scaled clusters, respectively.

Now we have the following two lemmas:

Lemma 6:

$$E(S_v) = C_h^{(1-\zeta)} E(S_h) \tag{4.14}$$

where $E(S_v)$ and $E(S_h)$ are the average task service times in a processing unit in a vertically scaled worker and a horizontally scaled worker, respectively.
Proof: In general, the mean task service time at a processing unit can be expressed as:

$$E(S) = \frac{E(W)}{v}, \tag{4.15}$$

where $E(W)$ is the task workload and v is the workload processing speed at a processing unit. Now, applying it to vertically and horizontally scaled workers, respectively, we have:

$$\begin{cases} E(S_h) = \frac{E(W_h)}{v_h} = \frac{E(W)}{C_h v_h}, & Horizontally\ scaled \\ E(S_v) = \frac{E(W_v)}{v_v} = \frac{E(W)}{v_v}, & Vertically\ scaled \end{cases} \tag{4.16}$$

So we have:

$$E(S_v) = \frac{E(S_h)C_h v_h}{v_v} = \frac{E(S_h)C_h v_h}{C_h^{\zeta} v_h} = C_h^{(1-\zeta)} E(S_h) \tag{4.17}$$

□

Lemma 7:

$$\rho_v = C_h^{(1-\zeta)} \rho_h \tag{4.18}$$

where ρ_v and ρ_h are the loads for vertically scaled and horizontally scaled clusters, respectively.
Proof: The load can be generally expressed as:

$$\rho = \lambda E(S). \tag{4.19}$$

where λ and $E(S)$ are average task arrival rate and average task service time, respectively. As explained in Section 2, the task arrival rates at both vertically and horizontally scaled clusters are the same. By multiplying both side of the equation in Lemma 6, we arrive at the equation in Lemma 7. □

With the above two lemmas, now we have the following result.

Theorem II: For $\zeta \geq 1$, vertical scaling outperforms horizontal scaling regardless of query fanout, k_f, and load, and for $\zeta < 1$, there is a crossover load, $\rho'_c(k_f) < 1$, a function of k_f, such that

$$\begin{cases} x_p^h(E(S_h), \rho_h, C_h k_f) = x_p^v(E(S_v), \rho_v, k_f), \ \rho_v = C_h^{(1-\zeta)} \rho_h = \rho'_c(k_f) \\ x_p^h(E(S_h), \rho_h, C_h k_f) < x_p^v(E(S_v), \rho_v, k_f), \ \rho_v = C_h^{(1-\zeta)} \rho_h > \rho'_c(k_f) \\ x_p^h(E(S_h), \rho_h, C_h k_f) > x_p^v(E(S_v), \rho_v, k_f), \ \rho_v = C_h^{(1-\zeta)} \rho_h < \rho'_c(k_f) \end{cases} \tag{4.20}$$

Proof: First, according to Eq. 4.13, $\zeta \geq 1$ means that vertical scaling has aggregate task workload processing power equal to or greater than horizontal scaling. Meanwhile, horizontal scaling incurs C_h times larger query fanout than vertical scaling, resulting in heavier tailed query distribution. Consequently, if $\zeta \geq 1$, vertical scaling outperforms horizontal scaling at any k_f and load.

Now, consider setting ζ to be smaller than one, i.e., vertical scaling has aggregate task workload processing power smaller than horizontal scaling. According to Eq. 4.18, $\rho_v > \rho_h$. So as the load increases, eventually, the query tail latency for vertical scaling will grow heavy enough to outweigh the heaviness of the query tail latency caused by C_h times larger fanout for horizontal scaling, i.e., a crossover load $\rho'_c(k_f)$ (< 1) exists for any given k_f. Moreover, $\rho'_c(k_f)$ reduces with ζ. □

Theorem II above clearly indicates that the query fanout plays an important role in determining which scaling solution should be adopted, vertical or horizontal.

The model developed so far is a unified one covering all the design and configuration options in Table I. and works for any task response time distribution, $F_T(t)$.

4.2.3 Performance Bounds

In this section, we consider a specific G/G/c task server model (note that G/G/1 is a special case of G/G/c by letting c=1) that can provide performance upper bounds. Specifically, we consider M/M/c queuing server, i.e., the case where the query/task arrival process is Poisson and the task service time distribution is exponential. While the Poisson arrival process, a non-bursty arrival process, is consider to be a reasonable model for query arrival processes in practice [29], the exponential service time distribution is considered much lighter tailed than the actual ones for datacenter applications [18]. Hence, the unified model with M/M/c task queuing servers is expected to provide performance upper bounds for DUS in general.

$F_T(t)$ for M/M/c queuing server is readily available [29] and can be written as follows,

$$F_T(t) = 1 - P(s > t) \tag{4.21}$$

and,

$$P(s > t) = \begin{cases} \frac{II_w}{1-c(1-\rho)} e^{-c(1-\rho)\mu t} + (1 - \frac{II_w}{1-c(1-\rho)})e^{-\mu t} & c(1-\rho) \neq 1 \\ (II_w t\mu + 1)e^{-\mu t} & c(1-\rho) = 1 \end{cases} \tag{4.22}$$

where μ $(= \frac{1}{E[S]})$ is the average task service rate, and,

$$II_w = \frac{(c\rho)^c}{c!}((1-\rho)\sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!})^{-1} \tag{4.23}$$

By letting $c = 1$, we have, for M/M/1 queue:

$$P(s > t) = e^{-(1-\rho)\mu t}. \tag{4.24}$$

With the above expression for $F_T(t)$ as input to the unified model, the performance bounds can then be derived for all the design and configuration options in Table I. The only parameter that needs to be measured is the mean task service rate, μ, or equivalently, the mean task service time, $E[S] = \mu^{-1}$. μ's for both scale-up (i.e., μ_u and scale-out (i.e., μ_o) workers are also needed to estimate the scaling factor δ in Eq. 4.8. The measurement cost is low, as $E[S]$ can be estimated as the average of a number of task service time samples, which can be collected with a single run of a task flow at a task server at any given load, assuming that the task service time is independent of the task queue length.

In particular, for a cluster in which scale-up workers are in use, i.e., they are modeled by M/M/1 queues, we have the following lemma:

Lemma 8: For a cluster with M/M/1 workers, the query tail-to-mean ratio, $\gamma_{tm} = \frac{x_p}{E[S]}$, is given by:

$$\gamma_{tm} = -\frac{1}{1-\rho} ln(1 - \sqrt[k_f]{\frac{p}{100}}). \tag{4.25}$$

or equivalently, the load is given by,

$$\rho = 1 + \frac{1}{\gamma_{tm}} ln(1 - \sqrt[k_f]{\frac{p}{100}}) \tag{4.26}$$

Proof: With Eq. 4.21, 4.24, 5.1 and 5.2 the p-th percentile tail latency can be written as:

$$x_p = G^{-1}(\frac{p}{100}) = -\frac{E(S)}{1-\rho} ln(1 - \sqrt[k_f]{\frac{p}{100}}) \tag{4.27}$$

we arrive at Eq. 4.25 by dividing both sides by $E(S)$. Solving this equation for ρ, we have Eq. 4.26. □

The above results may be used to estimate the achievable query tail-latency SLO lower bound at a given target query throughput/load or the query throughput/load upper bound at a given query tail-latency SLO for a cluster with scale-up workers. Similar performance bounds can also be estimated numerically for a cluster with scale-out workers (M/M/c), albeit the closed-form results are not available,

Following Theorem II, We also have the following result,

Corollary I: When $\zeta < 1$, the crossover load, $\rho_c'(k_f)$, for a cluster with M/M/1 fork nodes can be written as:

$$\rho_c'(k_f) = \frac{\frac{1}{c_h^{1-\zeta}} B(k_f, p) - 1}{B(k_f, p) - 1}, \tag{4.28}$$

where,

$$B(k_f, p) = \frac{ln(1 - \sqrt[c_h k_f]{\frac{p}{100}})}{ln(1 - \sqrt[k_f]{\frac{p}{100}})} > 1. \tag{4.29}$$

It can be easily verified that from Corollary I, $\rho_c'(K_f) \leq 1$ if only if $\zeta \leq 1$, which agrees with the general findings in Theorem II.

4.2.4 Task Response time distribution: $F_T(t)$

$F_T(t)$ cannot be easily obtained for $G/G/c$ queuing servers, other than $M/M/c$. A brute-force solution is to direct measure $F_T(t)$ at each load of interest (note that $F_T(t)$ is a function of load). For example, construct $F_T(t)$ in the form of a histogram using a large number of task response time samples as input, collected by running the task server at different loads of interest. The drawback of this approach is that it incurs high measurement cost, as it has to perform as many runs of experiment as the number of loads of interest and each run must be long enough to allow a sufficient number of samples to be collected in order to capture the tail portion of the distribution, critical for the prediction of the query tail latency performance. Instead of using this brute-force solution, in what follows, we propose a grey-box solution with the smallest possible measurement cost.

we start with a black-box solution [87, 88] that approximates $F_T(t)$ in general as a generalized exponential distribution function[49]

$$F_T(t) = (1 - e^{-t/\beta})^{\alpha} \tag{4.30}$$

where α and β are the shape and scale parameters, respectively. These parameters can be uniquely determined, as long as the mean, $E[T]$, and variance, $V[T]$, of the task response time are given. Namely, they are solvable from the following equations,

$$E[T] = \beta[\psi(\alpha + 1) - \psi(1)], \tag{4.31}$$

$$V[T] = \beta^2[\psi'(1) - \psi'(\alpha + 1)] \tag{4.32}$$

where $\psi(.)$ and its derivatives are digamma and polygamma functions, respectively.

The above modeling technique is found to be able to predict query tail performance within 20% errors for a wide range of task queuing servers of practical interests [87, 88]. This technique is a black-box solution because it is based on $E[T]$ and $V[T]$, which can be measured by treating the task queuing system as a black box, i.e., using the measured task response time samples as input. The needed number of samples for the estimation of $E[T]$ and $V[T]$ is expected to be smaller than that for the estimation of $F_T(t)$ directly. However, as both are load dependent, we still need separate runs of experiment at different loads of interest.

In this paper, we further reduce the measurement cost by proposing a grey-box solution based on the above technique. Specifically, we make use of the following known results for M/G/c queue.

Let Q, S, and T denote the random variables for task waiting time, service time, and response time, respectively. Assuming that Q and S are independent random variables, the mean and variance of the task response time can be written as,

$$E[T] = E[Q] + E[S], \tag{4.33}$$

$$V[T] = V[Q] + V[S]. \tag{4.34}$$

Then the mean and variance of the task response time for the M/G/1 queue are given as follows[74, 2],

$$E[Q] = E[S](1 + \frac{\rho}{1-\rho} \cdot \frac{1+C^2}{2}) \tag{4.35}$$

$$V[Q] = E^2[Q] + \frac{\lambda E[S^3]}{3(1-\rho)} + E[S^2] - E^2[S] \tag{4.36}$$

where $E[S^k]$ is the kth moment of the service time and C is the coefficient of variation of the service time distribution, . which is a function of $E[S]$ and $E[S^2]$.

Furthermore, an approximate solution for the mean and variance of the task response time for the M/G/c queue is given as follows [118],

$$E[Q] = \frac{(C^2+1)}{2}\frac{\rho}{\lambda_c(1-\rho)}II_w(c,\rho) \tag{4.37}$$

$$V[Q] = \frac{E[S^\gamma]}{E[S]^\gamma}^{\frac{2}{(\gamma-1)}}(V_D[Q] - E_D^2[Q]) + E^2[Q], \tag{4.38}$$

where C, again, is the coefficient of variation of the service time distribution, $E_D[Q]$ and $V_D[Q]$ are the mean and variance of the waiting times of the corresponding M/D/c queuing systems, and $E[S^\gamma]$ is the γth moment of the task service time and $\gamma = \gamma(c,\rho) \leq 3$, which is given in a table format [118].

With the above results, all we need to know are service time moments no greater than 3, i.e., $E[S^k]$, for $k = 1, 2, 3$ and $E[S^\gamma]$, where $\gamma = \gamma(c,\rho) \leq 3$. These moments can be estimated with the task service time samples collected from a single run of the experiment, the same as the case for M/M/c, i.e., the lowest possible measurement cost. Then we have $E[T]$ and $V[T]$ for both M/G/1 and M/G/c queues by plugging those moments into the above formulae. Finally, by solving α and β from Eqs. 4.32 and 4.31 and plugging them into Eq. 4.30, we find $F_T(t)$ for both M/G/1 and M/G/c queues.

Although the above grey-box solution assumes that the query/task arrival process is Poisson, our numerical results in the next section demonstrate that it can predict query tail latency with fairly high accuracy even for highly bursty arrival processes, such as the Pareto arrival process, especially in the high load region where the resource allocation is desired.

4.3 Numerical Analysis

In this section, we first verify the accuracy of the proposed models by simulation. Second, we conduct numerical analysis of the scale-up versus scale-out crossover load at the worker level and vertical versus horizontal crossover load at the cluster level.

4.3.1 Model accuracy evaluation

To check how accurate our model is for both the black box and grey box with measure mean and variance of task response time under every different load and one round mean and variance of task service time measurement, respectively. We run simulation tests for clusters with or without tail cutting or redundant issues and compare them with the numerical results in terms of the 99th percentile tail latency. The accuracy of the model is measured by the following relative error between the tail latency from the proposed model, x_p, and the one measured from simulation, x_s.

$$errorRate = \frac{x_p - x_s}{x_s}\% \tag{4.39}$$

where x_p and x_s are the tails obtained numerically and by simulation, respectively.

We evaluated both regular task service time with Exponential distribution and the heavy-tailed task service time distributions Weibull, and Truncated Pareto distribution as M/G/1 and M/G/c queuing model in another word grey box:

- Weibull distribution [16]:

$$F(t) = 1 - exp[-(t/\beta)^{\alpha}]\; t \leq 0, \tag{4.40}$$

  Here α and β are shape and scale parameters, respectively. We set $\alpha = 0.6848$ and $\beta = 3.2630$, so the mean service time is again 4.22ms.

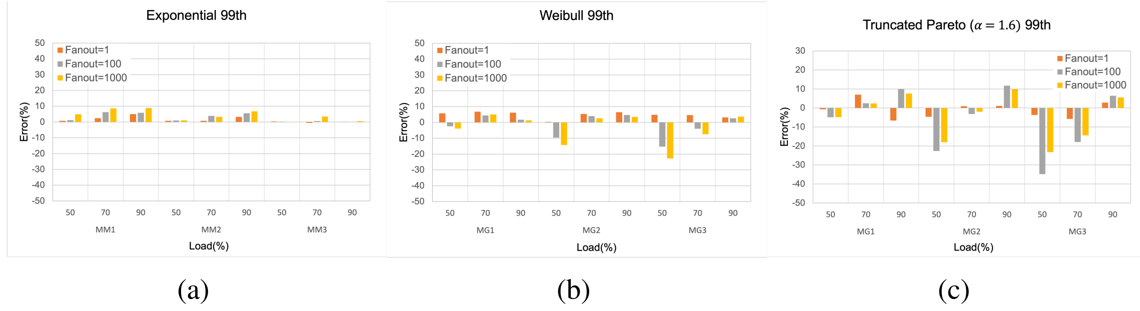
- Truncated Pareto distribution [4, 84, 87], with same mean service time 4.22ms. The CDF is:

$$F(t) = \begin{cases} \frac{1-(L/t)^{\alpha}}{1-(L/H)^{\alpha}} & 0 \leq L \leq t \leq H, \\ 0 & otherwise, \end{cases} \tag{4.41}$$

  where α is a shape parameter, $L = 2.023ms$ and $H = 276.6ms$ are the lower and upper bounds for the service time, respectively. We set $\alpha = 2.0119$ which is the worse case. We also set $\alpha = 1.6 < 2$ consider as a more practice case [42] with $L = 1.87ms$ and $H = 90ms$

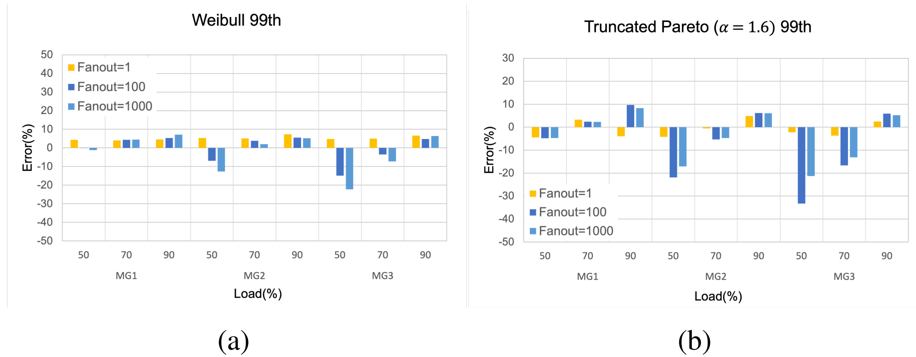
Note that with Exponential distribution which has a general solution, it transfers the M/G/1 and M/G/c queue to M/M/1 and M/M/c queue, respectively. The result for Exponential distribution with grey box is shown in figure 4.3a the error rate when $k_f = 1$ is less than 5% and the overall error rate of 99th percentile tail latency is less than 10%.

With the grey box way to measure the mean and variance service time and applying them to equation 4.30 to 4.38, the result is shown in Figure 4.3b and 4.3c. The error rate when $k_f = 1$ is less than 7% and 20% for Weibull and Pareto distribution, respectively. Note that when estimating the scale-up and scale-out crossover we only consider the $k_f = 1$ case. In this case, it proves that our model provides high accuracy with a grey box approach for worker-level scaling. The overall error rate is less than 23% for the Weibull distribution and for the Truncate Pareto distribution, this overall error rate is less than 20% at a higher load. Moreover, we assume the i.i.d case, the response time distribution function equation 5.1 is approximated due to the power of fanout k_f. This means with the fanout k_f increase there might be more error involved or neutralizing the error in the model. This explains why we can see the model first over-estimate and then underestimate or the error rate is keeping increasing or decreasing.



(a) (b) (c)

Figure 4.3: grey box error rate of 99th percentile tail latency, (a) Exponential (a) Weibull, and (c) Truncated Pareto

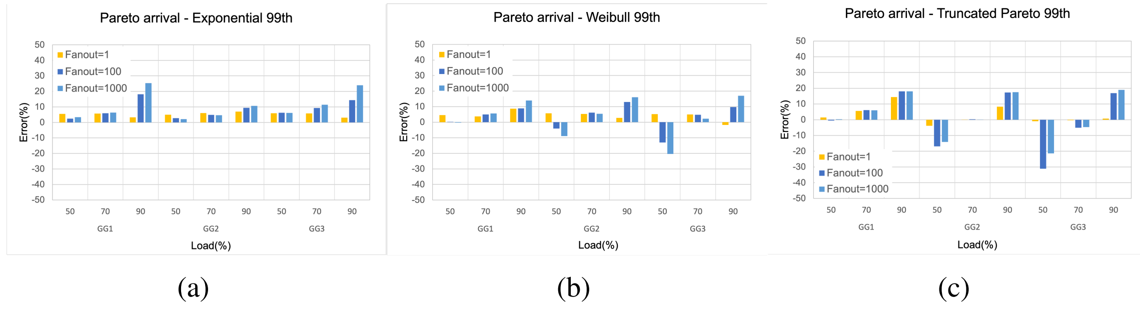


(a) (b)

Figure 4.4: Black box error rate of 99th percentile tail latency, (a) Weibull, (b) Truncated Pareto

Another thing that needs to pay attention is that we mentioned the high load's overall error rate is less than 20%, which matches the result shown in paper [87] that our model performs better at high load regions(78% to 95%). Moreover, in the real-world case, most of them target high load. The sensitivity analysis showed [87] that 20% error rate translates into about 7% error in load prediction, meaning that if our models are used to aid the resource provisioning, they may lead to at most 7% resource over-provisioning at high load regions. We also did the error evaluation with the black box approach which directly measured the mean and variance of task response time with every specific load and applied them to equation 4.30 to 4.32. Figure 4.4 shows the error rate of 99th percentile tail latency with Weibull, and Truncated Pareto distribution. As we can see the overall accuracy performs almost the same. This means that the grey box model we use has high accuracy. In other words, the grey box is good enough with low measurement complexity. So, in the rest evaluation for additional cutting tail design, we consider the grey box approach.

Besides the accuracy with different measurement values. There is one more thing that cannot be ignored, that is the grey box assumes the arrival rate is Poisson arrival. In



(a) (b) (c)

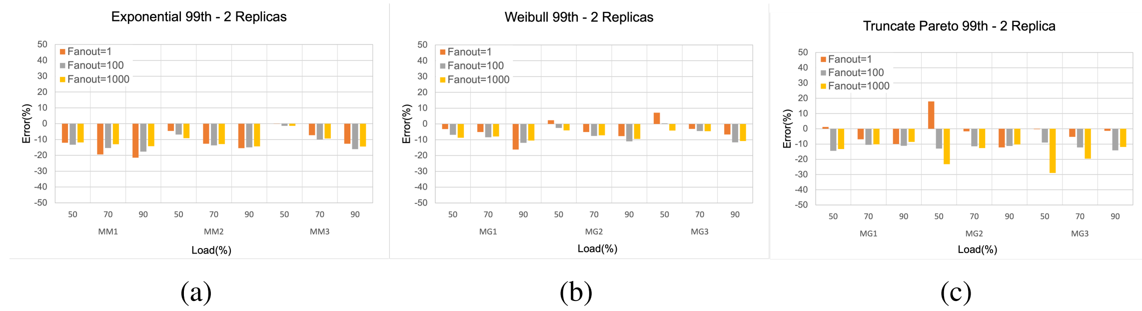
Figure 4.5: Black box error rate with Pareto arrival process, (a) Exponential (a) Weibull, and (c) Truncated Pareto

practice, the arrival rate can be the burstiness of request arrivals. We use a burstier arrival process, the Pareto arrival process[4]. The CDF of the Pareto distribution is

$$F(\alpha, t) = \begin{cases} 1 - (\frac{t_m}{t})^{\alpha} & t > t_m \\ 0 & otherwise, \end{cases} \tag{4.42}$$

where t_m is the minimum time interval, used as a tuning knob to adjust the load. α determines the variance of the arrival time interval, a measure of burtiness, which is less than 2 in practice [42]. We set $\alpha = 1.6$ to capture heavier bursty arrivals and the queue will be considered as G/G/c or G/G/1 queue which we use black box to test the accuracy. We run the simulation with Weibull and Truncate Pareto distribution as service time distribution with load from 10% to 90%. In the meantime, we collect the mean and variance of task response time to get the α and β values with eq. 4.31 and 4.32 then apply to eq. 4.30 to get the 99th tail latency. The error rate result is shown in Figure 4.5. With both Poisson and Pareto arrival processing, fanout $k_f = 1, 100, 1000$, different $c = 1, 2, 3$ numbers represented scale-up and scale-out as Figure 4.5a, 4.5b, and 4.5c show the overall error rate is less than 20% at higher load. This means the burtiness arrival rate doesn't affect the accuracy of our model much. So, in the rest of the evaluation we only consider the Poisson arrival distribution.

The above evaluation is for basic design without the redundant issue and tail-cutting. However, in the real world, the cutting tail configuration is needed to satisfy the tight tail latency requirement. Also, people want to achieve a high load to improve resource utilization at a low tail-to-mean ratio γ_{tm} as we mentioned in the previous Section. For example, when $\gamma_{tm} = 40$ with Weibull service time distribution the maximum load it can achieve without redundant issues is 70% while with the redundant issue the maximum load increases to about 90%. This clearly shows that if we want to achieve high resource utilization the



(a) (b) (c)

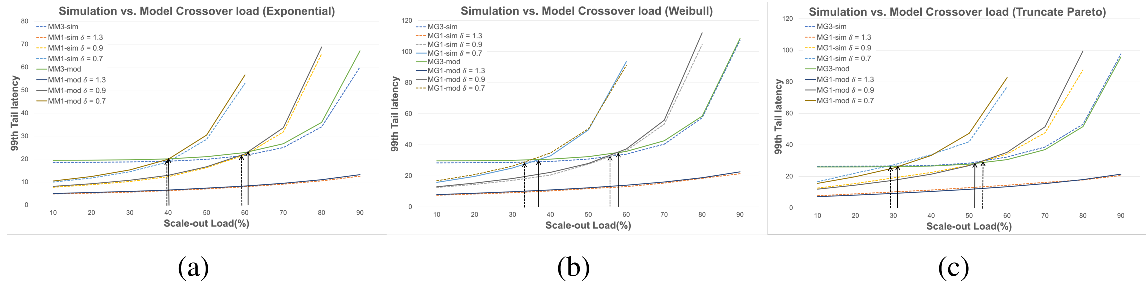
Figure 4.6: grey box with redundant issue error rate of 99th percentile tail latency, (a) Exponential (a) Weibull, and (c) Truncated Pareto

cutting tail configuration is needed. Under this condition, the following evaluation result shows that our model can give high accuracy as well when adding additional cutting tail configuration: redundant issue, and tail-cutting.

In the rest of this section, we evaluated how our model performed with redundant issues and tail-cutting under grey box approaches. We first give the redundant issue configuration evaluation result. Figure 4.6 shows the 99th percentile tail latency error rate with Exponential, Weibull, and Truncated Pareto with 2 replicas. As we can see the overall error rate for both is less than 20% at higher load, even for low load(50%0 is less than 30%. This gives us an idea that if the budget allows, adding the configuration of the redundant issue is a good way to have better performance in the meantime our models can provide high-accuracy predicted tail results that translate into about 7% resource overprovisioning. Moreover, our model is the best case but we set our simulation to be a worst-case without any cancel policy. By comparing the best and worst case the overall error rate is less than 20% we can see that any other better solution will definitely reduce the error rate. It again proves that our model provides highly accurate predict.

Tail-cutting configuration is focused on handling the outlier, such as VM failure or network connecting issue which happened very few. So, we set the cutting threshold to be 2-3 times the tail. The result shows that there is no significant difference with the regular design and the overall error rate is less than 20% at higher load, again transferring to about 7% error for resource provisioning.

In conclusion, our unified model with simple one-round measurement independent of load can provide high-accuracy performance evaluation connected to different system designs and configurations.



(a) (b) (c)

Figure 4.7: Scale-up VS Scale-out 99-th percentile tail latency with different arrival rate (load): (a) Exponential (a) Weibull, and (c) Truncated Pareto

4.3.2 Crossover analysis

In this section, we give the crossover load prediction result with our grey box model and compare it with the simulation result at both the worker level and cluster level.

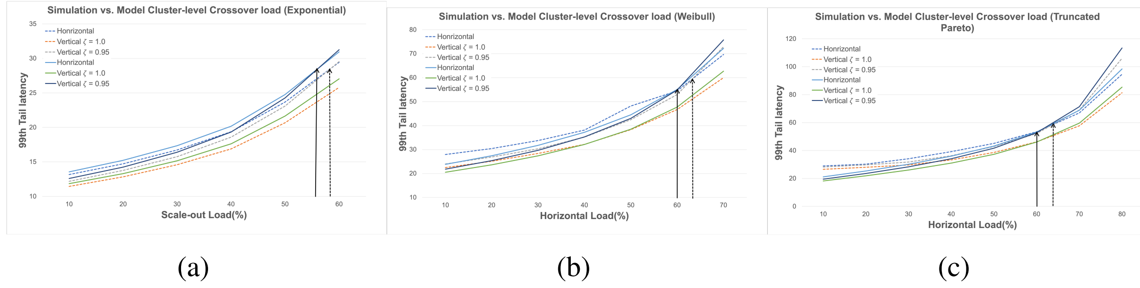
Scale-up versus scale-out At the worker level the crossover load, ρ_c, is independent of query fanout, k_f, so we conduct $k_f = 1$ at the worker-level analysis.

We first consider the M/M/1, M/G/1 queue for the scale-up worker versus the M/M/c, M/G/c queue for the scale-out worker with the Exponential, Weibull and Truncated Pareto service time distribution. We set the number of processing units in a scale-out worker, $C_o = 3$ and $p = 99$th percentile, and scaling factor $\delta = 0.7, 0.9, 1.3$. As shown in Figure 4.7a, 4.7b 4.7c, the arrow points to the crossover point, and the dotted line is the simulation result, the solid line is the model result, as it shows that the smaller δ is, the smaller ρ_c will be. When $\delta > 1$ the scale up is better than the scale out, agreeing with Theorem I and our postulation. Table 4.7 shows the crossover point error rate by comparing the simulation and model which is less than 5%.

Vertical versus Horizontal At the cluster level, we set $c_h = 3$ and $k_f = 20$ for the vertical cluster. To show the crossover point, we evaluate the 99th-percentile tail latency with $\zeta = 1.0, 0.95$. The result is shown in Figure 4.8. Again, the arrow points to the crossover point, the dotted line is the simulation result, and the solid line is the model result. And, table 4.3 gives a clear look at crossover load ρ_c. The overall error rate when predicting the crossover is less than 5%. In contrast, people can use our model to make a more effective and accurate resource provisioning decision.

4.4 Related Work

First, queuing models have been widely used for performance modeling of distributed computing in datacenters and clouds. Mary and Saravanan [82] model the cloud



(a) (b) (c)

Figure 4.8: Vertical VS Horizontal 99-th percentile tail latency with different arrival rate (load): (a) Exponential (a) Weibull, and (c) Truncated Pareto

Table 4.3: Crossover load (Simulation vs Model)

Distribution	Worker-level($\delta = 0.9$) Sim(%)	Mod(%)	Error rate(%)	Cluster-level($\zeta = 0.95$) Sim(%)	Mod(%)	Error rate(%)
Exp	60.62	61.57	1.57	59.40	54.60	8.08
Wei	55.63	57.69	4.17	64.32	61.16	-4.91
TPar	53.73	52.98	-1.40	65.03	62.82	-3.40

center as an M/G/1 queue server with query response time and waiting time distributions obtained. Chang et al.[21] develop an approximate analytical model to evaluate the performance of an M/G/m/m + K queuing system and obtain the mean queue length, the mean response time, and the blocking probability. Bai et al.[14] construct a complex queuing model with a single queue and multiple heterogeneous servers to evaluate the performance of heterogeneous data centers. They analyze the mean response time, the mean waiting time, and other performance indicators. They also mention that the configuration of server clusters may impact the performance of the system. Khazaei et al[72] propose an analytical model based on the M/G/m queue for the performance evaluation of a cloud computing environment. They examine the effects of various parameters, such as arrival rate, the size of the task, and virtualization degree on the task rejection probability and delay. Fakhrolmobasheri et al[40] propose an analytical model based on an M/M/1 queue for the performance evaluation of Infrastructure-as-a-Service (IaaS) cloud systems, also considering failure/repair behavior of virtual machine monitors and virtual machines. Hanini and El Kafhali [50] propose an analytical model based on M/M/c queue, derive performance parameter expressions to estimate the loss probability, mean number of requests in the system, and mean requests delay, while varying the incoming request arrival rate. Hellemans et al.[55] provide a numerical method to assess the performance of workload dependent policies including many replication policies, as well as strategies for fork-join systems.

Scully et al.[104] study the optimal mean response time policy in the M/G/k queue. Wang et al[130] derive stability, queueing probability, and the transient analysis of the number of jobs in the system based on the M/M/c queue.

However, The above works are focused on the job mean response time, rather than the tail latency, hence more applicable to batch applications than user-facing applications. Moreover, the underlying single queuing server structure of these works renders them incapable of capturing the fanout nature of the DUS workloads, which must be modeled with multiple queuing servers working in parallel with barrier, e.g., Fork-Join queuing network models.

For the sake of completeness, in what follows, we briefly review the related work on different design and configuration aspects covered in our paper, most of which, however, are focused on scaling analysis of specific real systems with limited configuration options, rather than performance modeling of various possible system designs and configurations, including scale-out, scale-up, vertical scaling, horizontal scaling, redundant issues, and tail cutting.

Vertical and horizontal scaling Dawoud et al. [31] compare vertical and horizontal scaling and use a simple threshold-based controller for adapting the resources. CloudScale [107] uses resource requirement predictions to scale a VM vertically. Yazdanov et al. [137] use an auto-regressive prediction model to predict the resource requirements without considering the application performance as a result of scaling. Lakew et al. [76] and Farokhi et al.[41] propose significantly faster average response time models for vertically scaled resource allocation. Spinner et al.[110] propose a performance model to ensure that the application performance meets the user-defined SLO efficiently by runtime vertical scaling of individual virtual machines, and they [109] also propose a proactive vertical memory approach to adjust settings associated with application memory management during memory reconfiguration. Rossi et al[100] propose Reinforcement Learning (RL) solutions for controlling the horizontal and vertical elasticity of container-based applications with the goal to increase the flexibility to cope with varying workloads. All of these works are mainly focused on the online scaling of real systems and some of them do so without considering the performance.

Scale-out and Scale-up TwoSpot[133] is a platform built for scaling out and it makes the scaling decision depending on the application load. Ferdman et al[43] use performance counters to analyze the micro-architectural behavior of a wide range of scale-out workloads. Ramanathan et al.[95] present a performance model for predicting job completion time and the Hadoop MapReduce jobs execution time in a private cloud environment using the scale-out strategy. Hwang et al[58, 59] evaluate both scale-out and scale-up strategies to satisfy production services and they argue that the choice of scale-up or scale-out so-

lution should be made primarily by the workload patterns and resource utilization rates required. Michael et al.[85] investigate the behavior of two competing approaches to parallelism, scale-up and scale-out, in an emerging search application. Their result shows that a scale-out strategy can be the key to good performance even on a scale-up machine. However, their evaluation is performed on special hardware. Appuswamy et al.[11] claim that a single "scale-up" server can process each of these jobs and do as well or better than a cluster in terms of performance, cost, power, and server density. Moreover, they propose that the correct decision of scale-out or scale-up depends on job size, job characteristics, and pricing.

Redundant issue and tail cutting Many works have been devoted to reducing tail latency with redundant task issues or tail cutting.

Some works demonstrate that redundant issues can help improve performance in real-world applications. Vulimiri et al.[126] demonstrate that redundancy is an effective general technique to achieve low latency in networked systems. Dean et al.[32] note that Google's big table services use redundant issues in order to improve latency. Joshi et al.[64, 66] analyze how redundant issues affects the latency and the cost of computing time under fork-join structure and propose a general redundancy strategy for an arbitrary service time distribution. Zoolander [113] uses redundant accesses to mask outlier response times. C3 [117] develops an adaptive replica selection mechanism to reduce the latency tail in the presence of service-time fluctuations in the system. Ayesta et al.[13] present a unifying analysis for redundancy systems with different cancel policies with exponentially distributed service times.

Works that allow partial results to be returned to fulfill a query fall into the category of tail cutting techniques include [54, 61, 138, 10, 140]. All these solutions achieve more predictable query performance at the cost of possible loss of partial results.

In summary, the existing performance models and works concerning specific systems have limited scope, covering only certain aspects of the system design and configuration options concerning the current work.

4.5 Conclusions

This paper provides mathematical models to characterize the performance for data-intensive, user-facing applications (DUSes). The proposed models provide a direct connection between tail latency, throughput, load, and resource demand under a range of system configurations and design choices. We first propose models for both worker-level and cluster-level scaling and different configurations. Then we describe the performance scaling models that establish a connection between resource scaling and performance for differ-

ent systems. We also prove that under certain resource scaling conditions, there is a cross-over load beyond which the horizontal scaling/scale-out outperforms vertical scaling/scale-up at the cluster/worker level. We also discussed how our models may be applied to aid resource planning for datacenter clusters to support DUSes.

CHAPTER 5

Initial Resource Provisioning Plan Tool

5.1 Introduction

With the rapid evolution of cloud computing, traditional on-premise computing is being surpassed. According to Foundry's recent Cloud Computing Study (2022), a staggering 84% of organizations already have some of their applications or computing infrastructure in the cloud. Furthermore, the study reveals that nearly three-quarters of the 850 surveyed organizations default to cloud-based services. Consequently, many organizations are actively migrating their existing workflows and applications to the cloud. As user numbers increase and data access expands, services are becoming more data-intensive. These services, known as Data-intensive User-facing services, include web searching, banking apps, digital marketing, online social networking, and edge-based crowd-sensing for emergency response.

For data-intensive user-oriented services, time-sensitive factors, especially query tail latency, are widely recognized as critical requirements and have a huge impact on user experience, application adoption, customer satisfaction, and ultimately business revenue and success. Moreover, there is a wide range of possible cluster design and configuration options and different options may lead to vastly different query tail latency and throughput performance. We identify some widely adopted design and configuration choices, in terms of, scale-up or scale-out, with or without redundant task issues [126] at the worker level; and vertical scaling or horizontal scaling with or without tail cutting [61] at the cluster level. This leads to a total of 16 distinct design and configuration options. Although, related issues of cloud computing resource planning and query tail latency optimization have been made: some of these studies [131] focus on developing online optimization algorithms and scheduling strategies to achieve low-latency services under high load conditions, others based on queuing models [130, 104], they have limited scope, focusing on the comparative study of a few design and configuration options.

In addition, cost-effectiveness is also an essential consideration in cloud computing. Average cloud server prices range from $400 per month (one server) to $15,000 per month (entire back-office infrastructure). This only covers the cost of server rental and does not include other expenses such as maintenance and operations. We refer to the purchase price plus operating costs as the total cost of ownership (TCO) is an essential indicator

that enterprises need to consider in cloud computing. Methods to reduce cloud computing costs include optimizing resource utilization, adopting elastic scaling strategies, and flexible pricing models. Effectively reducing operating costs and improving resource utilization efficiency is critical for users to make cost-effective plans when deploying services to the cloud.

We proposed an offline initial resource provisioning plan tool based on the offline performance model that provides a direct connection between cluster resource demand, query tail-latency SLO and throughput for all design and configuration options to predict the tail latency and resource utilization with different system designs and configurations so that users can use the predicted result combined with their special requirement to have an efficient initial plan with as less as further changes after the system is deployed.

The rest of this paper is organized as follows.

5.2 Model

The models we used to build our tool for different system designs and configurations are described in Section 4.2 of Chapter 4.

We put some important formulas here for easy reading.

Lemma 1: The query response time distribution $G(t)$ can be approximately written as,

$$G(t) \approx F_T(t)^{k_f} \tag{5.1}$$

Lemma 2: The pth-percentile query tail latency of x_p is given by,

$$x_p = G^{-1}(\frac{p}{100}) \tag{5.2}$$

Theorem I: For $\delta > 1$, the scale-up worker outperforms the scale-out worker in terms of the tail-latency performance, in the entire load range of $\rho_{up} \in [0, 1]$. and for $\delta < 1$,There is a cross-over load, $\rho_c < 1$, independent of query fanout k_f, such that,

$$\begin{cases} x_p^{out}(\mu_o, \rho_{out}, k_f) = x_p^{up}(\mu_u, \rho_{up}, k_f), & \rho_{up} = C_o^{(1-\delta)}\rho_{out} = \rho_c \\ x_p^{out}(\mu_o, \rho_{out}, k_f) > x_p^{up}(\mu_u, \rho_{up}, k_f), & \rho_{up} = C_o^{(1-\delta)}\rho_{out} < \rho_c \\ x_p^{out}(\mu_o, \rho_{out}, k_f) < x_p^{up}(\mu_u, \rho_{up}, k_f), & \rho_{up} = C_o^{(1-\delta)}\rho_{out} > \rho_c \end{cases} \tag{5.3}$$

where $x_p^{out}(\mu_o, \rho_{out}, k_f)$ and $x_p^{up}(\mu_u, \rho_{up}, k_f)$ are the pth-percentile query tail latencies for the scale-out and scale-up clusters, respectively.

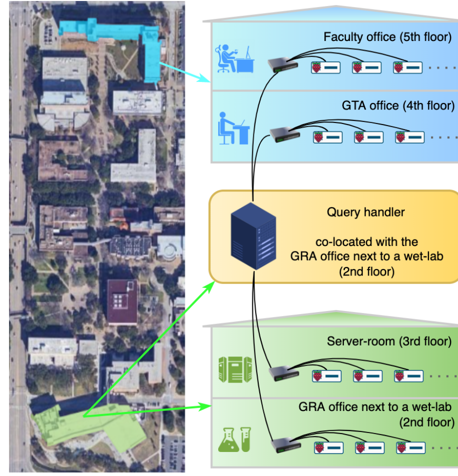


Figure 5.1: Locations (blue and green colored buildings) of the edge nodes and the query handler in the SaS testbed.

Theorem II: For $\zeta \geq 1$, vertical scaling outperforms horizontal scaling regardless of query fanout, k_f, and load, and for $\zeta < 1$, there is a crossover load, $\rho'_c(k_f) < 1$, a function of k_f, such that

$$\begin{cases} x_p^h(E(S_h), \rho_h, C_h k_f) = x_p^v(E(S_v), \rho_v, k_f), & \rho_v = C_h^{(1-\zeta)} \rho_h = \rho'_c(k_f) \\ x_p^h(E(S_h), \rho_h, C_h k_f) < x_p^v(E(S_v), \rho_v, k_f), & \rho_v = C_h^{(1-\zeta)} \rho_h > \rho'_c(k_f) \\ x_p^h(E(S_h), \rho_h, C_h k_f) > x_p^v(E(S_v), \rho_v, k_f), & \rho_v = C_h^{(1-\zeta)} \rho_h < \rho'_c(k_f) \end{cases} \tag{5.4}$$

5.3 Methodology

5.3.1 System description

We illustrate our system with our on-campus SaS testbed. The testbed is currently composed of four clusters of Raspberry Pi edge nodes, located in four rooms in two buildings, including a server room and a Graduate Research Assistant (GRA) office next to a wet lab in one building, and a faculty office and a Graduate Teaching Assistant (GTA) office in another building as shown in Figure 5.1. In each cluster, there are 8 Raspberry Pi devices, serving as edge nodes, and connected to the Internet through an Ethernet switch. As mentioned, we identify some widely adopted design and configuration choices from the worker to the cluster levels. To cover every choice our tools workflow is shown in Figure 5.2 which gives an overview of how the tool works. As Figure 5.2shows,

Step 1: The user who will rent resources from the resource provider needs to provide some basic information and requirements of their services, such as average arrival rate λ, mean

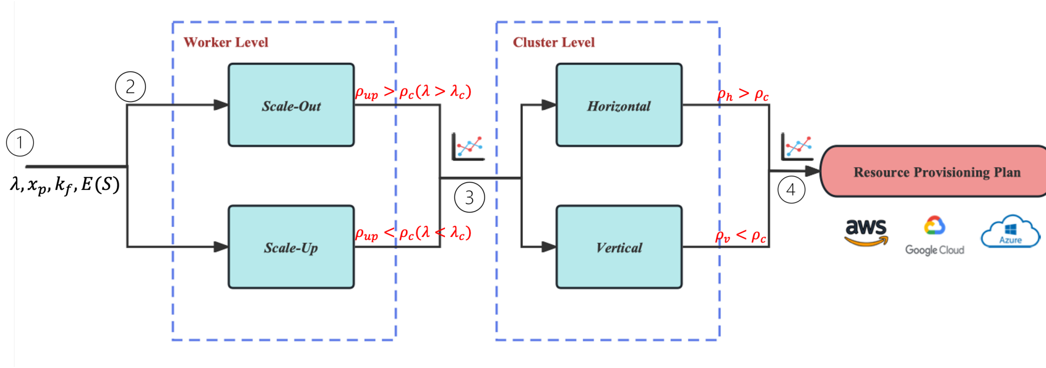


Figure 5.2: Overview

service time $E(S)$, fanout k_f, and tail latency SLO x_p. More input detail will give in Section 5.3.2;

Step 2: Decide the worker-level scaling design. With pre-measurement $delta$ of the different type processing unit, if $delta < 1$ with **Theorem I** we know that there will be a cross-over load which will be given by our model so that people know they should choose scale-out or scale-up at the worker-level based on their tail latency requirement. At this step, the output will be a figure that shows both scale-out and scale-up tail latency versus system load. Users can use this figure with the tail latency to figure out it is in which area and related to which scaling method to make the decision. More detail can be found in Section 5.4. Now, let's assume that the input load is smaller than the cross-over load then we choose the scale-up for each worker;

Step 3: with the previous selection, first of all, each worker in the cluster should apply a scale-up design. Then, apply fanout k_f and ζ to our cluster-level crossover model we can make the cluster-level scaling strategy selection. Again, if $\zeta < 1$ there is a cross-over load based on **Theorem II**. Same as worker-level, it will also generate a guild figure for the user to relate the tail latency and scaling method. Again, let's assume the Horizontal scaling can provide better performance since we found that $\rho_v < \rho_c$;

Step 4: Redo Step *1-3* with the redundant issue and cutting tail configuration if necessary(based on user's additional requires);

Step 5: Combine all the selection with comparison to make the final initial resource allocation.

5.3.2 Input Variables and Requirements

The input parameter can be classified into two types: variables from measurement; and predefined variables which are based on the user application and requirement.

Worker level		Cluster level	
Variable	Description	Variable	Description
μ_u	Average task service rate for scale-up worker	$E(S_v)$	Mean task service time for vertically scaled cluster
μ_o	Average task service rate for scale-out worker	$E(S_h)$	Mean task service time for horizontally scaled cluster
$E(R_u)$	Mean task response time for scale-up worker	$E(R_v)$	Mean task response time for vertically scaled cluster
$E(R_o)$	Mean task response time for scale-out worker	$E(R_h)$	Mean task response time for horizontally scaled cluster
$V(R_u)$	Variance task response time for scale-up worker	$V(R_v)$	Variance task response time for vertically scaled cluster
$V(R_o)$	Variance task response time for scale-out worker	$V(R_h)$	Variance task response time for horizontally scaled cluster
δ	Scaling factor at worker level	ζ	Scaling factor at cluster level

Table 5.1: Measurement variables

Predefined variables The predefined variables as following include the variables directly from the application and other outside factors such as budget, fault-tolerant, reliability, and so on.

- C_o: Number of processing units in scale-out worker
- C_h: Number of horizontally scaled workers
- k_f: Average query fanout
- λ: Average query arrival rate
- X_p^{SLO}: p-th percentile tail-latency SLO requirement
- other user requirements: reliability, budget.

Measurement Variables: The measurement variables mean that the user needs to run their DUSes on a basic processing unit. The input measurement variables are shown in Table 5.1 and we give a detailed explanation of each as follows. Noted, not all the measurement variables are required in the table, the user needs to collect whatever they need based on the model they use.

White box model: The white box means that task service time distribution is already known and follows kind specific distribution, such as, Exponential, Weibull, Pareto, and so on. This means that we can use the queuing model like M/M/c or M/G/c to get the query response time distribution as Eq. 4.8 and calculate the pth percentile tail latency. In this way, if the task service time distribution is Exponential distribution (M/M/c queuing model), the μ_u and μ_o are the variables needed at the worker-level and apply them to Eq. 4.21 - 4.24 can get the δ value. Same, at the cluster level, the $E(S_v)$ and $E(S_v)$ are needed then we have the ζ value by applying to Eq. **??**. Otherwise, for any other task service time distributions we can use the M/G/c queuing model. And, there are two different cases: First, the γ $(\gamma \leq 3)$ moment can be calculated with a known equation, then the measurement variable is the same as the Exponential distribution; Second, the moment calculate equation is unknown, then up to 3 moments of the task service time need to be measured at both worker-level and cluster level as Eq. **??** - 4.38 shows.

Black box model: Different from the white box, the black box means that the task service time distribution of the DUSes is unknown. In this case, we need to measure all of the variables in Table 5.1 to get the δ, ζ, and query response time distribution function with Eq. 4.30 to 4.32 to calculate the pth percentile tail latency.

5.4 Case Study

In this section, we will give a step-by-step case study that shows how to use our tool to do the initial resource provisioning plan. In the end, we deploy the system with the plan and verified it by running the application on our testbed system.

5.4.1 Predefined requirements and variables

Our predefined variables are as follows:

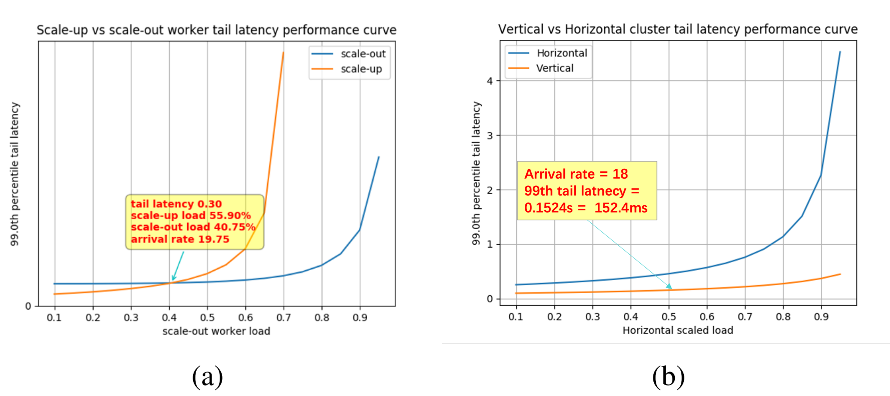
- C_o: 3
- C_h: 2
- k_f: 15
- λ: 18
- X_p^{SLO}: 200ms
- other user requirements: budget limited

5.4.2 Worker level input variables

First of all, we need to collect the input variables. Based on the testbed we discuss in Section 5.3.1 and our application. We decided to use the white box model and trade the task service time following the Exponential distribution. At the worker level, we choose the basic processing unit (we use "OUT" to represent it) in the scale-out worker with 30% of 1 core CPU resource and each contains 3 ($C_o = 3$) processing units. Equally, the scale-up worker is one big processing unit (we use "UP" to represent it) with 90% of 1 core CPU resource. Then we run the application on each of the "OUT", and "UP" to collect μ_u, μ_o and use those two values to calculate the δ values. All the variables we measured are shown in Table 5.2 which shows $\delta = 0.712 < 1$ and means there should be a crossover point between scale-out and scale-up worker. By inputting these worker-level measured values into our tool we have the following Figure 5.3a as output. As the figure shows, the crossover point is clearly shown in the figure that it happened when the arrival rate is 19.75. This means if the application arrival rate is less than 19.75 the scale-up worker provide better tail performance, otherwise the scale-out worker is better.

Worker level		Cluster level	
Variable	Value	Variable	Value
μ_u	35.34	$E(S_v)$	11.97
μ_o	16.16	$E(S_h)$	28.30
δ	0.712	ζ	1.242

Table 5.2: Measurement variables



(a) (b)

Figure 5.3: Tool output

5.4.3 Cluster level input variables

At the cluster level, 2 of the wimpy worker (we use "VAR" to represent it) in the horizontally scaling cluster is equal to 1 brawny worker (we use "HON" to represent it) in the vertically scaling cluster which means the $C_h = 2$. At the worker level the crossover arrival rate $\lambda_c = 19.75$, and both arrival rate is less than the crossover arrival rate, so we choose scale-up worker as each worker's scaling design. Then we run the application on both the "VAR" and "HON" to collect $E(S_v)$, $E(S_h)$, respectively. Then use those values to calculate the ζ values. The variables we measured are shown in Table 5.2 as well which shows $\zeta = 1.242 > 1$ and means that there is no crossover happening the vertically scaling cluster is always better than the horizontally scaling cluster. By inputting these cluster-level measured values into our tool we have the following Figure 5.3b. In the meantime, we run our application based on the configuration we chose to verify our model. We set each worker as a scale-up worker and use a total of 15 nodes in our testbed to build the vertically scaling system. The 99th percentile tail latency results with $\lambda = 18$ is 153.3ms which is very close to the result our tool gave 152.4ms. Moreover, it meets the predefined requirement. There is one more thing that needs to be paid attention to the

request tail latency is $200ms$, from Figure 5.3b it can also be satisfied even if we choose a horizontally scaling cluster. In this case, the user can use our output figures combined with their requirements to make the decision very flexible.

This case study is to give our users an idea of how to use our initial resource provisioning tool to make their specific plan. More case studies such as the AWS EC2 resource plan will be considered as future work.

5.5 Conclusion

This chapter provides a resource provisioning tool that is built on top of a set of mathematical models. The mathematical models provide a direct connection between a set of performance metrics such as tail latency and throughput, and resource demand under different system configurations and design choices. With our tool, users can easily make a flexible plan with lower complexity measurements. We also give a case study with our on-campus testbed to show some ideas of how our tool can provide. For future work, we will extend our tool to more application scenarios such as AWS EC2, Google Cloud, and so on.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Contributions

The contributions of this dissertation can be summarized as follows:

- We propose Pigeon, a distributed hierarchical job scheduler for datacenters. In Pigeon, workers are divided into groups. Each group has a master worker which centrally manages all the tasks handled by the group. Weighted fair queuing is used to provide priority service differentiation between tasks of short jobs and tasks of long jobs. A small portion of workers in each group are reserved to serve short job tasks only. The ability of each master in managing its group resources centrally makes Pigeon highly effective in scheduling heterogeneous jobs. Evaluation via theoretical analysis, trace-driven simulations, and a prototype implementation shows that Pigeon significantly outperforms 30X to 150X with other schedulers Sparrow up to 150X at 90th percentile tail latency for short jobs(user-facing services).
- We propose TailGuard based on a Tail-latency-SLO-and-Fanout-aware Earliest-Deadline-First Queuing policy (TF-EDFQ) for task queuing at individual task servers the query tasks are fanned out to. With the task queuing deadline for each task being derived based on both query tail latency SLO and query fanout, TailGuard takes an essential first step towards achieving the design objective. A query admission control scheme is also developed to provide tail latency SLO guarantee in resource shortages. TailGuard is evaluated against First-In-First-Out (FIFO) task queuing, task PRIority Queuing (PRIQ) and Tail-latency-SLO-aware EDFQ (T-EDFQ) policies by both simulation and testing in the Amazon EC2 cloud. The experiment results demonstrate that TailGuard can improve resource utilization by up to 80%.
- We develop queuing models that provide a direct connection between cluster resource demand, query tail-latency SLO and throughput for all design and configuration options. We derive the maximum sustainable cluster loads at different query tail-latency-to-mean ratios for different design and configuration options; And we prove that under certain resource scaling conditions, there is a worker-level cross-over load, below (above) which the scale-up (scale-out) workers outperform scale-out (scale-up) ones, independent of query fanout, and there is a cluster-level cross-over load, a function of query fanout, below (above) which the vertical-scaling (horizontal-

scaling) outperforms horizontal-scaling (vertical-scaling). We perform a comprehensive test of the accuracy of the proposed models in predicting the DUS performance by simulation. The result shows that the overall accuracy of our model is under 30% which is about 10% resource over-provisioning at high load region.

- We build a tail latency-aware initial resource provisioning plan tool on top of our performance model. Users can use this tool with simple input effectively make their initial resource plan, thereby making as less as future changes after deployment on the cloud.

6.2 Future Work

The resource provisioning tool described in Section 5 is implemented only for the off-line initial resource provisioning plan, and gives one use case with the on-campus test-bed. Our goal is to build a resource provisioning tool for DUSes users with the cloud. With this goal, we have the following future work targeted:

- Evaluated the existing tool with different real cloud service providers such as AWS, and GCP with real-cloud VMs based on the real Data-intensive user-facing application. Based on the plan build a completed large scale such as a 100 - 200 nodes cluster system and run the real DUSes to verify our plan;
- Involve budget into our model to have a tool that can give not only a good performance plan but also consider the net profit. With the budget and cost involved the tool can be used by the tenant to maximize their income while providing better service to their customer.
- Apply our model to online resource provisioning. Nowadays, many DUSes are already deployed on the cloud, but with the increase of users and data, they need to be scaled. So, how to scale more efficiently and make it as less effective as the running system is very important.

REFERENCES

[1] Kubernetes (K3s). https://k3s.io/.

[2] M/G/c Queue. https://en.wikipedia.org/wiki/M/G/k_queue.

[3] Order statistic. https://en.wikipedia.org/wiki/Order_statistic.

[4] Pareto distribution. https://en.wikipedia.org/wiki/Pareto_distribution.

[5] Poisson Distribution. https://en.wikipedia.org/wiki/Poisson-distribution.

[6] Pollack's rule. https://en.wikipedia.org/wiki/Pollack′s_rule.

[7] Storage: How Tail Latency Impacts Customer-Facing Applications. https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications.

[8] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of ACM SIGCOMM*, 2018.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, and N. McKeown. pFabric: Minimal Near-Optimal Datacenter Transport. In *ACM SIGCOMM*, 2013.

[10] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, 2001.

[11] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–13, 2013.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of ACM SIGMETRICS*, 2012.

[13] U. Ayesta, T. Bodas, and I. M. Verloop. On a unifying product form framework for redundancy models. *ACM SIGMETRICS Performance Evaluation Review*, 46(3):80–81, 2019.

[14] W.-H. Bai, J.-Q. Xi, J.-X. Zhu, and S.-W. Huang. Performance analysis of heterogeneous data centers in cloud computing using a complex queuing model. *Mathematical Problems in Engineering*, 2015, 2015.

[15] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. RobinHood: Tail Latency- Aware Caching - Dynamically Reallocating from Cache-Rich to Cache-Poor. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[16] G. Bolch, S. Greiner, H. d. Meer, and K. S. Trivedi. Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science . *Wiley-Interscience*, 2006.

[17] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of OSDI*, 2014.

[18] O. Boxma and B. Zwart. Tails in scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):13–20, 2007.

[19] J. Brutlag. Speed matters for google web search. In *Google*, 2009.

[20] B. Cai, K. Li, L. Zhao, and R. Zhang. Less Provisioning: A Hybrid Resource Scaling Engine for Long-running Services with Tail Latency Guarantees. *IEEE Transactions on Cloud Computing*, 2020.

[21] X. Chang, B. Wang, J. K. Muppala, and J. Liu. Modeling active virtual machines on iaas clouds using an m/g/m/m+ k queue. *IEEE Transactions on services computing*, 9(3):408–420, 2014.

[22] W. Chen, J. Rao, and X. Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. In *Proceedings of USENIX Annual Technical Conference*, 2017.

[23] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, C. Reynold, and H. Cui. Achieving Low Tail-latency and High Scalability for Serializable Transactions in Edge Computing. In *Proceedings of ACM Eurosys*, 2021.

[24] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of VLDB Endowment*, 2012.

[25] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of MASCOTS*, 2011.

[26] Y. Cheng, A. Anwar, and X. Duan. Analyzing Alibaba's Co-located Datacenter Workloads. In *Proceedings of IEEE International Conference on Big Data (BIGDATA)*, 2018.

[27] S. Cho, A. Carter, J. Ehrlich, and J. A. Jan. Moolle: Fan-out Control for Scalable Distributed Data Stores. In *Proceedings of IEEE International Conference on Data Engineering*, 2016.

[28] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of ACM SIGCOMM*, 2015.

[29] R. B. Cooper. *Introduction to Queueing Theory*. North Holland, 1981.

[30] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[31] W. Dawoud, I. Takouna, and C. Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In *Global Trends in Computing and Communication Systems: 4th International Conference, ObCom 2011, Vellore, TN, India, December 9-11, 2011. Proceedings, Part I*, pages 11–25. Springer, 2012.

[32] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[33] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), 2013.

[34] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: divide and stick to your probes. In *Proceedings of ACM Symposium on Clod Computing (SOCC)*, 2016.

[35] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates . In *Proceedings of ACM Symposium on Clod Computing (SOCC)*, 2018.

[36] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2015.

[37] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. *ACM SIGPLAN Notices*, (4), 2013.

[38] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *ACM SIGPLAN Notices*, 49(4), 2014.

[39] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of ACM SIGCOMM*, 2014.

[40] S. Fakhrolmobasheri, E. Ataie, and A. Movaghar. Modeling and evaluation of power-aware software rejuvenation in cloud systems. *Algorithms*, 11(10):160, 2018.

[41] S. Farokhi, P. Jamshidi, D. Lucanin, and I. Brandic. Performance-based vertical memory elasticity. In *2015 IEEE International Conference on Autonomic Computing*, pages 151–152. IEEE, 2015.

[42] A. G. Fayoumi. Performance evaluation of a cloud based load balancer severing pareto traffic. *Journal of Theoretical and Applied Information Technology*, 32(1):28–34, 2011.

[43] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[44] A. D. Fergusin, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of EuroSys*, 2012.

[45] A. S. Foundation. Hadoop: Yarn federation, 2018.

[46] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmanent: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of USENIX Symposium on Operating System Design (OSDI)*, 2016.

[47] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can Jump Them! . In *Proceedings of ACM NSDI*, 2015.

[48] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of IEEE/ACM International Conference on Quality of Service (IWQoS)*, 2019.

[49] R. D. Gupta and D. Kundu. Theory & methods: Generalized exponential distributions. *Australian & New Zealand Journal of Statistics*, 41(2):173–188, 1999.

[50] M. Hanini and S. El Kafhali. Cloud computing performance evaluation under dynamic resource utilization and traffic control. In *Proceedings of the 2nd international Conference on Big Data, Cloud and Applications*, pages 1–6, 2017.

[51] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. *ACM SIGPLAN Notices*, (4), 2015.

[52] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[53] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling Interactive Services with Partial Execution. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2012.

[54] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[55] T. Hellemans, T. Bodas, and B. Van Houdt. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–35, 2019.

[56] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI*, 2011.

[57] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of SoCC*, 2011.

[58] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on parallel and distributed systems*, 27(1):130–143, 2015.

[59] K. Hwang, Y. Shi, and X. Bai. Scale-out vs. scale-up techniques for cloud performance and productivity. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 763–768. IEEE, 2014.

[60] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of SOSP*, 2012.

[61] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.

[62] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cos. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *Proceedings of ACM ASPLOS*, 2016.

[63] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive Parallelization: Taming Tail Latencies in Web Search. In *Proceedings of the ACM SIGIR*, 2014.

[64] G. Joshi, E. Soljanin, and G. Wornell. Queues with redundancy: Latency-cost analysis. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):54–56, 2015.

[65] G. Joshi, E. Soljanin, and G. Wornell. Efficient Redundancy Techniques for Latency Reduction in Cloud Systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2(2), 2017.

[66] G. Joshi, E. Soljanin, and G. Wornell. Efficient redundancy techniques for latency reduction in cloud systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 2(2):1–30, 2017.

[67] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[68] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications. In *Proceedings of IEEE International Workshop/Symposium on Workload Characterization*, 2014.

[69] K. Karanasos, S. Rao, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2015.

[70] H. Kasture, D. B. Bartolini, N. B. Beckmann, and D. Sanchez. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[71] H. Kasture and D. Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *Proceedings of IEEE International Workshop/Symposium on Worklod Characterization (IISWA)*, 2016.

[72] H. Khazaei, J. Misic, and V. B. Misic. Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems. *IEEE Transactions on parallel and distributed systems*, 23(5):936–943, 2011.

[73] M. Khelghatdoust and V. Gramolim. Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queuess. In *Proceedings of International Conference on Parallel and Distributed Computing*, 2018.

[74] L. Kleinrock. Theory, volume 1, queueing systems, 1975.

[75] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. *ACM SUGARCH Computer Architecture News*, 45(1), 2017.

[76] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Towards faster response time models for vertical elasticity. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 560–565. IEEE, 2014.

[77] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2014.

[78] N. Li, H. Jiang, D. Feng, and S. Zhang. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceeding of the European Conference on Computer Systems (EuroSys)*, 2016.

[79] X. Liang, M. Nguyen, and H. Che. Wimpy or brawny cores: A throughput perspective. *Journal of Parallel and Distributed Computing*, 73(10):1351–1361, 2013.

[80] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of ACM ISCA*, 2015.

[81] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the Cloud: an Analysis on Alibaba Cluster Trace. In *Proceedings of IEEE International Conference on Big Data (BIGDATA)*, 2017.

[82] N. A. B. Mary and K. Saravanan. Performance factors of cloud computing data centers using [(m/g/1):([infinity]/gdmodel)] queuing systems. *International Journal of Grid Computing & Applications*, 4(1):1, 2013.

[83] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[84] D. Meisner, J. Wu, and T. F. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 35–45. IEEE, 2012.

[85] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE international parallel and distributed processing symposium*, pages 1–8. IEEE, 2007.

[86] P. A. Misra, M. F. Borge, I. Goiri, A. R. Lebeck, W. Zwaenepoel, and R. Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the European Conference on Computer Systems (Eurosys)*, 2019.

[87] M. Nguyen, S. Alesawi, N. Li, H. Che, and H. Jiang. Forktail: A black-box fork-join tail latency prediction model for user-facing datacenter workloads. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 206–217, 2018.

[88] M. Nguyen, Z. Li, F. Duan, H. Che, and H. Jiang. The tail at scale: how to predict it? In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[89] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of USENIX NSDI*, 2013.

[90] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander. Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[91] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[92] C. Perera, A. Zaslavsky, and Georgakopoulos. Sensing as a service model for smart cities supported by Internet of Things. In *Wiley Transactions on Emerging Telecommunications Technologies*, 2013.

[93] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, H. Mars, and L. Tang. Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[94] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of ACM SOSP*, 2017.

[95] R. Ramanathan and B. Latha. Towards optimal resource provisioning for hadoop-mapreduce jobs using scale-out strategy and its performance analysis in private cloud environment. *Cluster Computing*, 22(6):14061–14071, 2019.

[96] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient Queue Management for Cluster Scheduling. In *Proceedings EroSys*, 2016.

[97] W. Reda, M. Canini, L. Suresh, D. Kostic, and S. Braithwaite. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of ACM Eurosys*, 2017.

[98] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, 2012.

[99] S. Rosenkrantz, H. Li, P. Enganti, Z. Li, L. Sun, Z. Wang, H. Che, and H. Jiang. JADE: Tail-Latency-SLO-Aware Job Scheduling for Sensing-as-a-Service. In *Proceedings of International Workshop on Cloud and Edge Computing, and Applications Management*, 2020.

[100] F. Rossi, M. Nardelli, and V. Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 329–338. IEEE, 2019.

[101] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of IEEE International Conference on Cloud Computing*, 2011.

[102] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes. Autopilot: Workload Autoscaling at Google. In *Proceedings of European Conference on Computer Systems (Eurosys)*, 2020.

[103] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*, 2013.

[104] Z. Scully, I. Grosof, and M. Harchol-Balter. The gittins policy is nearly optimal in the m/g/k under extremely general conditions. In *Abstract Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, pages 15–16, 2021.

[105] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[106] R. Sheldon. *Introduction to Probability Models*. Academic Press, 2014.

[107] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[108] X. Sheng, J. Tang, X. Xiao, and G. Xue. Sensing as a Service: Challenges, Solutions and Future Directions. *IEEE Sensors Journal*, 13(10), 2013.

[109] S. Spinner, N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, and R. Griffith. Proactive memory scaling of virtualized applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 277–284. IEEE, 2015.

[110] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith. Runtime vertical scaling of virtualized applications via online model estimation. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 157–166. IEEE, 2014.

[111] A. Sriraman and T. F. Wenisch. μSuite: A Benchmark Suite for Microservices. In *Proceedings of IEEE International Workshop/Symposium on Worklod Characterization (IISWA)*, 2018.

[112] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *Proceedings of International Conference on Autonomic Computimg (ICAC)*, 2013.

[113] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *ICAC*, volume 13, pages 265–277, 2013.

[114] R. S. Stutsman. Durabilit and Crash Recovery in Distributed In-Memory Storage Systems . In *Dissertation of Doctor Philosophy*, 1987.

[115] k. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *Proceedings of ACM European Conference on Computer systems (EuroSys)*, 2018.

[116] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceeding of USENIX NSDI*, 2015.

[117] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 513–527, 2015.

[118] Y. TAKAHASHI. An approximation formula for the mean waiting time of an m/g/c queue. *Journal of the Operations Research Society of Japan*, 20(3):150–163, 1977.

[119] H. Tan, S. Jiang, Y. Li, X. Li, C. Zhang, H. Zhenhua, and F. C. M. Lau. Joint Online Coflow Routing and Scheduling in Data Center Networks. *IEEE/ACM Transactions on Networkingl*, 27(50), 2019.

[120] A. Thrift. Apache thrift, 2017.

[121] H. Tian, Y. Zheng, and W. Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2019.

[122] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the Next Generation. In *Proceedings of European Conference on Computer Systems (Eurosys)*, 2020.

[123] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of EuroSys*, 2016.

[124] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-Aware Datacenter TCP (D^{2}TCP). In *ACM SIGCOMM*, 2012.

[125] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, 2013.

[126] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294, 2013.

[127] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low Latency via Redundancy. In *Proceedings of ACM CoNEXT*, 2013.

[128] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2012.

[129] D. Wang, G. Joshi, and G. Wornell. Using straggler replication to reduce latency in large-scale parallel computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):7–11, 2015.

[130] W. Wang, Q. Xie, and M. Harchol-Balter. Zero queueing for multi-server jobs. *ACM SIGMETRICS Performance Evaluation Review*, 49(1):13–14, 2021.

[131] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, and H. Jiang. Pigeon: an effective distributed, hierarchical datacenter job scheduler. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2019.

[132] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of ACM SIGCOMM*, 2011.

[133] A. Wolke and G. Meixner. Twospot: A cloud platform for scaling out web applications dynamically. In *European Conference on a Service-Based Internet*, pages 13–24. Springer, 2010.

[134] Y. Xia, R. Ren, H. Cai, A. V. Vasilakos, and Z. Lv. Daphne: A Flexible and Hybrid Scheduling Framework in Multi-Tenant Clusters. *IEEE Transactions on Network and Service Management*, 15(1), 2018.

[135] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[136] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. *ACM SIGARCH Computer Architecture News*, 41(3), 2020.

[137] L. Yazdanov and C. Fetzer. Vertical scaling for prioritized vms provisioning. In *2012 Second International Conference on Cloud and Green Computing*, pages 118–125. IEEE, 2012.

[138] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 63–72, 2015.

[139] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of EuroSys*, 2010.

[140] F. Zhang, S. Shi, H. Yan, and J.-R. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 371–380, 2010.

[141] T. Zhu, D. S. Berger, and M. Harchol-Balter. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2016.

[142] T. Zhu, M. A. Kozuch, and M. Harchol-Balter. WorloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2017.

[143] T. Zhu, A. Tumanov, M. a. Kozuch, M. Harchol-Balter, and G. R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

BIOGRAPHICAL STATEMENT

Huiyang Li was born in Taiyuan, Shanxi in 1992. She received her B.E. degrees in Electronic and Information Engineering from University of Science and Technology Beijing, Tianjin College, Tianjin, China, in 2014. She received her M.E. degree in Software Engineering from the University of Texas at Arlington in 2017. She received her Ph.D. degree in Computer Sciences from the University of Texas at Arlington in 2023. Her main areas of research interest are datacenter resource management and the theoretical foundation for both off-line and on-line resource allocation in the cloud.