Distributed In-Network Task Scheduling for Datacenters

by

Parham Yassini

B.Sc., Amirkabir University of Technology, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Parham Yassini 2021
SIMON FRASER UNIVERSITY
Fall 2021

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Parham Yassini**
Degree:	**Master of Science**
Thesis title:	**Distributed In-Network Task Scheduling for Datacenters**
Committee:	**Chair:** Ouldooz Baghban Karimi Lecturer, School of Computing Science **Mohamed Hefeeda** Supervisor Professor, School of Computing Science **Khaled Diab** Committee Member University Research Associate, School of Computing Science **Alaa Alameldeen** Examiner Associate Professor, School of Computing Science

Abstract

Scheduling latency-sensitive applications in large-scale datacenters is challenging. Current approaches use application-layer schedulers, which impose high overheads and result in long latencies. We present Saqr, the first in-network, datacenter-wide scheduler that supports short tasks with execution times in the order of tens of microseconds. Saqr introduces new network-level constructs and a distributed scheduling policy to enable network switches to efficiently schedule tasks within the network at line rate and with minimal latency. We implemented Saqr in a testbed with high-speed programmable switches and compared its performance against the state-of-the-art in-network scheduler (Racksched). Our results show that Saqr can reduce the tail response time by up to 85% and the processing load on switches by up to 2.5X compared to Racksched. In addition, we compared Saqr versus Racksched using large-scale simulations with diverse and dynamic workloads and our results show that Saqr substantially outperforms Racksched across all performance metrics.

Keywords: Scheduling, Datacenter, Programmable Dataplane

Dedication

To my beloved wife and my dear parents; the ones who have always been there for me unconditionally.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Mohamed Hefeeda for his immense efforts in guiding me along the path. His support, encouragements, suggestions, and scientific knowledge were critical in the completion of this thesis.

Also, I would like to express my deepest thanks to Dr. Khaled Diab for being an inspiration to me and mentoring me with patience and genuine care. I will always be grateful for all I have learned from him and for his help in all stages of this thesis. It would be impossible for me to finish this thesis without his invaluable guidance.

I am also thankful to the committee members Dr. Baghban Karimi as the chair and Dr. Alameldeen as the examiner.

Table of Contents

Declaration of Committee **ii**

Abstract **iii**

Dedication **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Figures **viii**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Thesis Contributions 2

1.3 Thesis Organization 3

2 Background, Related Work and Challenges **4**

2.1 Background 4

2.2 Related Work 7

2.3 Challenges of In-network Scheduling 8

3 Proposed in-Network Scheduling System **10**

3.1 Overview and Design Principles 10

3.2 Scheduling Tasks in the Network 13

3.2.1 Policy Overview 13

3.2.2 Scheduling Tasks to Idle Nodes 13

3.2.3 Scheduling Tasks to Busy Nodes 15

3.3 Distributing State Among Schedulers 18

3.3.1 Distributing Worker State to the Leaf Layer 18

3.3.2 Distributing Rack State to the Spine Layer 18

3.4 Handling Failures 21

3.5 Selecting Spine Schedulers 21

4 Implementation and Practical Considerations **23**
4.1 Implementation of Saqr . . . 23
4.2 Practical Considerations . . . 24

5 Evaluation in a Testbed **26**
5.1 Testbed Setup . . . 26
5.2 Experiments and Results . . . 28

6 Evaluation using Simulation **33**
6.1 Simulation Setup . . . 33
6.2 Comparison against the State-of-Art . . . 34
6.3 Analysis of Saqr . . . 35

7 Conclusions and Future Work **39**
7.1 Conclusions . . . 39
7.2 Future Work . . . 40

Bibliography **41**

List of Figures

Figure 2.1 Example of a multi-rooted Clos network topology. 5
Figure 2.2 Illustrative overview resource manager architectures. Dotted rectangles represent the worker state managed by each scheduler. 6
Figure 2.3 Example of a programmable switch architecture. 7

Figure 3.1 Overview of Saqr: The first distributed in-network task scheduler for datacenters. 12
Figure 3.2 Saqr header format. 14
Figure 3.3 Illustrative example of scheduling tasks to busy workers with selective state updates in Saqr. 16
Figure 3.4 Distributing idle state from a leaf to spine scheduler. 19
Figure 3.5 Distributing load information from leaf to spine schedulers. 20

Figure 5.1 Testbed setup. The hardware resources of a Tofino switch are partitioned to emulate one spine switch and four leaf switches. Multiple servers are connected to leaf switches. 27
Figure 5.2 Variations of Racksched. 27
Figure 5.3 Response time in the Uniform worker placement. 29
Figure 5.4 Response time in the Skewed worker placement. 29
Figure 5.5 Rate of state updates. 30
Figure 5.6 Fraction of resubmitted tasks. 31
Figure 5.7 Total packet processing overheads. 32

Figure 6.1 Response time for different workload distributions in our simulations. . . 34
Figure 6.2 Impact of worker placement. 35
Figure 6.3 Rate of state updates. 35
Figure 6.4 Break-down of Saqr performance benefits. 36
Figure 6.5 Impact of number of samples on scheduling decisions for different policies. Dotted line shows the performance of an *Oracle* scheduler. 37
Figure 6.6 Impact of spine switch failures on the applications. 38

Chapter 1

Introduction

Recent interactive, user-facing, applications running in multi-tenant datacenters have tight response time requirements. Examples of these applications include key-value stores [7, 6, 8], web search [5, 13], and function-as-a-service platforms [18, 29, 66]. Each one of these datacenter applications typically runs multiple *tasks* in parallel to complete a function or job. The average execution time of tasks in latency-sensitive applications is very short; it ranges from tens to hundreds of microseconds [45, 12]. Tasks are executed by computing resources in the datacenter, which are referred to as *workers*. Workers run on the servers of the datacenter. To execute a task, a worker needs to be chosen, which is the job of the *datacenter scheduler*.

Since a datacenter application runs multiple tasks in parallel, its response time is affected by the slowest task. This is referred to as the *tail response time*, which needs to be minimized for the application to meet its strict requirements. In addition, large-scale datacenters with tens of thousands of servers are expected to support millions of concurrent tasks. That is, datacenter schedulers are expected to support *high throughput*.

Traditional datacenter-wide schedulers run as application-layer processes [55, 72, 33, 24, 31]. This introduces significant network and processing delays, especially for latency-sensitive applications. In addition, these schedulers may not achieve the high throughput needed for modern large-scale datacenter. For example, in current datacenters, hundreds of millions of scheduling decisions need to be made each second. This scheduling workload requires a substantial amount of computing resources [72], and it is difficult to realize using traditional application-layer schedulers.

Unlike traditional, application-layer, datacenter schedulers, in-network schedulers make task scheduling decisions by the network switches themselves. While in-network schedulers have the potential to reduce the tail response time for applications and achieve high scheduling throughput, they are challenging to design. This is because of the stringent limitations imposed by the network switches. These limitations include a restricted programming model and small amounts of memory and processing resources on the switches. In this thesis, we address this challenge and design a scalable and efficient in-network datacenter scheduler.

1.1 Problem Statement

The problem we address in this thesis is designing an efficient task scheduler that scales to datacenters with large number of workers while supporting short latency-sensitive tasks. Designing scheduler for large-scale datacenters, is a challenging for multiple reasons. First, running short μs-scale tasks on a large number of workers requires high throughput of scheduling decisions. Second, meeting the strict latency requirement of such workloads requires efficient scheduling scheduling decisions that minimize the task waiting times. Third, to make effective scheduling decisions, the scheduler needs to keep track of the state of the available resources across the entire datacenter, which could impose substantial communication and processing overheads.

In summary, the problem considered in this thesis can be stated as follows. Design an in-network task scheduler that assigns each arriving task to one of the computing resources in the datacenter such that the average task response time is minimized, while imposing minimal overheads on the network switches and achieving high scheduling throughput.

1.2 Thesis Contributions

We propose Saqr,[1] the first datacenter-wide in-network task scheduler for multi-tenant datacenters. Unlike traditional schedulers, Saqr *offloads* the scheduling functionality to distributed switches in the network, which enables efficient and fast scheduling of latency-sensitive tasks. Despite its potential benefits, designing in-network scheduling is challenging as it needs to balance between the scheduling quality (i.e., the tail response time and throughput) and the communication and processing overheads imposed on switches. Saqr addresses this trade-off by distributing the scheduling workload across different switches in the datacenter.

In designing Saqr, we introduce a new scheduling policy that does not require queuing incoming tasks at switches, which enables task scheduling at line rate. We also present multiple ideas and data structures to efficiently realize the proposed scheduling policy in modern programmable switches, which have a restricted programming model and limited memory resources. In addition, we propose simple mechanisms to distribute the load information among in-network schedulers while balancing between the freshness of load values and communication and processing overheads on switches. Furthermore, We present efficient methods to handle switch and worker failures.

We implement the proposed scheduler in a testbed using a high-end programmable switch, which is logically divided into multiple switches to emulate a representative part of a datacenter network. We compare Saqr against Racksched [76], the state-of-art in-network scheduler. Our experimental results show that Saqr can reduce the tail response time by up to 85%, increase the throughput by up to 2.17X, and reduce the processing load on switches by up to 2.5X, compared to Racksched. In addition, we analyze the performance of various components of Saqr and show

[1] Saqr is the Arabic word for falcon, the fastest known animal.

its robustness to failures using simulations. We also compare Saqr versus Racksched in large-scale simulation setups, using datacenter topologies with more than 27K hosts and thousands of switches. The simulation setups consider diverse and dynamic scheduling workloads. Our simulation results show that Saqr substantially outperforms Racksched across all performance metrics.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents a brief background on datacenter scheduling and in-network computing. It also summarizes the related work in the literature and challenges of designing in-network datacenter task schedulers. Chapter 3 presents the details of the proposed solution. Chapter 4 presents our implementation of the task scheduler in the P4 language, which is the stat-of-the-art programming language for programmable network switches. It also describes various practical considerations for Saqr. Chapter 5 describes the setup of our testbed and presents our experimental results. And Chapter 6 evaluates the proposed solution using large-scale simulations. Finally, Chapter 7 concludes the thesis and describes potential directions for future work.

Chapter 2

Background, Related Work and Challenges

This chapter presents a brief background on datacenter networks, task scheduling in datacenters and the modern programmable switches. Next, it summarizes the related works in the literature. Finally, it discusses the challenges of realizing task scheduling inside the network.

2.1 Background

Datacenter Network Topology. Layered tree model is used in many modern datacenter network designs. Particularly, a widely deployed architecture in the datacenters is the multi-rooted Clos tree topology [9], which consists of core, spine, and leaf layers. Figure 2.1 shows an example of such architecture. Each leaf switch (also known as top-of-rack switch) is connected to a rack of servers. Spine switches handle inter-rack routing. In this architecture, multiple paths can exist between a given source and destination server. The topology also addresses the challenges of providing high bisection bandwidth and supporting large number of servers.

Scheduling in Datacenters. Resources in datacenters are commonly managed by systems called *cluster managers*. Cluster managers are responsible for sharing the resources among different applications and scheduling incoming applications [64, 35]. As illustrated in Figure 2.2, the scheduling architecture in cluster managers can be divided into two main categories : (i) *Monolithic* where decisions are serialized and one scheduler makes all of the scheduling decisions [27]. (ii) *Two-level* where a resource manager offers the resources to multiple schedulers [35, 71], and each scheduler allocates these resources to the arriving tasks. The schedulers operate independently and in parallel. The two-level design addresses the scalability issues of handling scheduling decisions at high rates and provides lower latency per decision. Fine-grained task schedulers, such as [55] and [76], focus on the second level of scheduling in the two-level design and address the problem of assigning tasks to the *allocated resources*.

Ideally, the resource allocator should provide applications with enough resources to satisfy the performance requirements [28], and the fine-grained task scheduler should assign the tasks to the

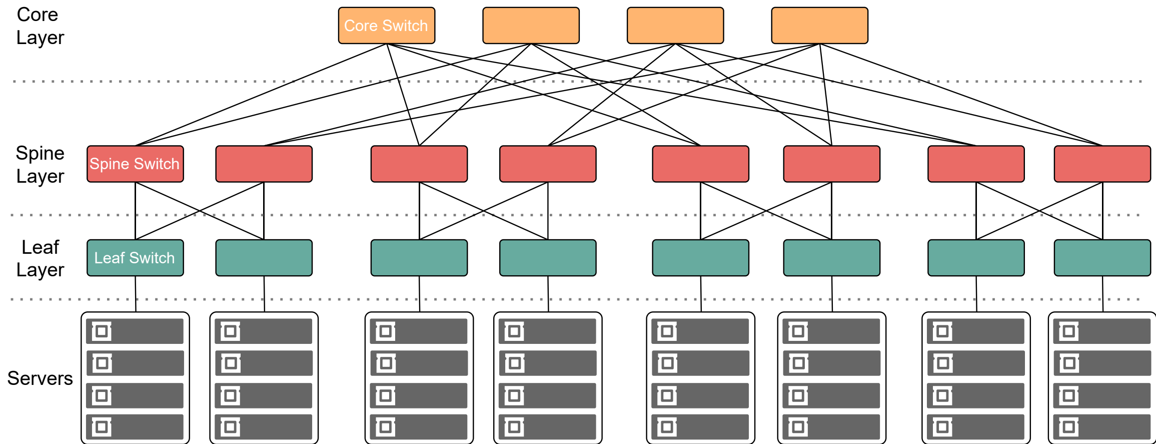


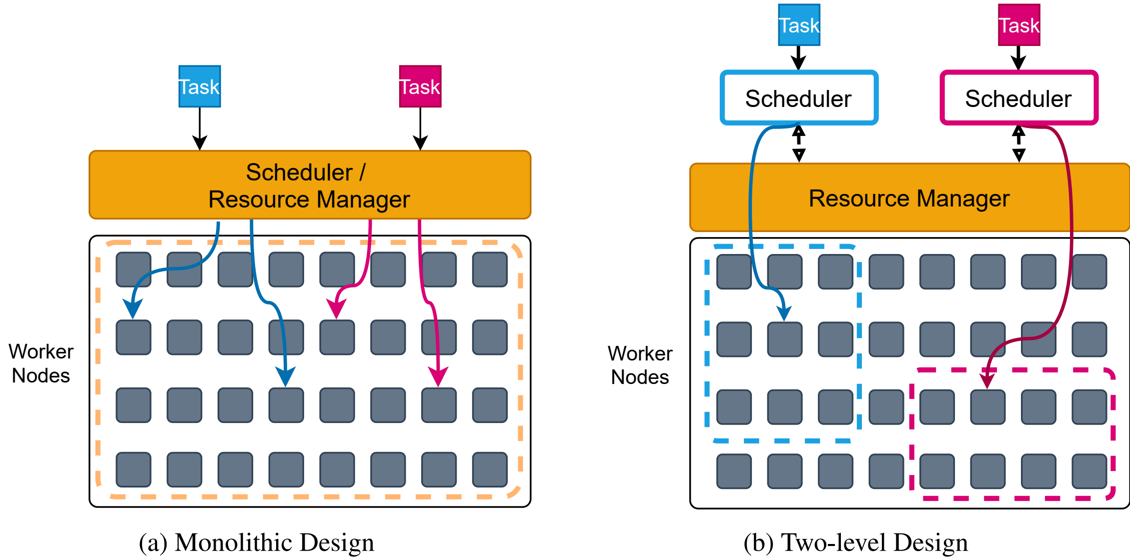
Figure 2.1: Example of a multi-rooted Clos network topology.

available resources to minimize the response time to meet the service level objectives (SLO). Scheduler inefficiency, however, forces datacenter operators to over-provision the resources in order to provide a predictable performance [25, 51], which results in underutilized clusters. For example, a study on a cluster that serves user-facing services in Twitter showed that the aggregate cluster CPU utilization is below 20% [27].

To support the increasing demand and with the end of Moore's law, there is a recent trend to build highly parallel scalable applications [38, 45, 29] with shorter task service times. For example, modern online services already involve thousands of nodes to serve a single customer query [61, 12]. As the tasks become shorter and the number of workers increase, the schedulers need to provide higher throughput and lower latency for making decisions. For example, considering a cluster with 20K workers. Given a task mean execution time of 100 μs, the system needs to make 200M scheduling decisions per second and to handle around the same number of packets for processing the state update messages.

Programmable Data Planes. Networks consist of *data plane* and *control plane* components. The data plane performs the packet forwarding, and the control plane is the part that gives instructions to the forwarding components to process the packets (e.g., routing tables). In traditional switches, the data plane is realized as a fixed and specialized hardware to provide high speed while the control plane runs on top of general purpose CPUs to enable more flexibility. However, recent advancements [17] have enabled the data-planes to be programmable while achieving forwarding capacity at scale of few Tbps. That is, modern programmable switches offer a great potential for offloading parts of the applications to the network. This enables the applications to achieve higher throughput and lower latencies [59].

Figure 2.3 shows an example of the programmable switch architecture. At a high level, there are three main programmable blocks in this architecture: (i) Parser, which provides the state machine for extracting the header field values, (ii) Match-Action pipeline, which contains memory and ALU units to perform the operations on the header fields, and (iii) Deparser, which re-assembles the



(a) Monolithic Design

(b) Two-level Design

Figure 2.2: Illustrative overview resource manager architectures. Dotted rectangles represent the worker state managed by each scheduler.

packet and calculated headers. The main part of processing logic is realized using the match-action pipeline which is composed of multiple processing stages [17]. Each stage has memory and computing resources. Transferring data between stages is done via the metadata bus, where the result of computations from one stage can be passed to subsequent ones.

Defining the behaviour of the mentioned programmable blocks is done using domain-specific languages such as P4 [16]. At compile time, each part of the packet processing logic is mapped to the resources of a certain stage of the pipeline, and each stage accesses a pre-allocated section of the SRAM abstracted as register arrays.

Scaling out beyond a single rack. Recent works, e.g., [76, 43, 42], showed the potential of programmable switches for realizing high throughput and low-latency scheduling. However, they are mostly designed to schedule tasks *within individual racks*, and they do not scale to the whole datacenter. Although some applications running in datacenters may not need more cores than the available in one rack, there are many practical scenarios where applications require and/or benefit from execution on cores across different racks. For example, running applications across racks in different fault domains improves their fault-tolerance and availability [15, 3, 1]. This is especially important for latency-sensitive applications, since most of them are user-facing in production services. A recent study from Facebook [2] indicates that the traffic of many latency-sensitive applications is mostly not rack-local. In addition, in public datacenters, it is not uncommon that tenant's Virtual Machines (VMs) are placed on different racks by the placement algorithm due to unavailable resources at the time [36] or for improved fault tolerance. Therefore, there is a need to run tasks across racks in the datacenter.

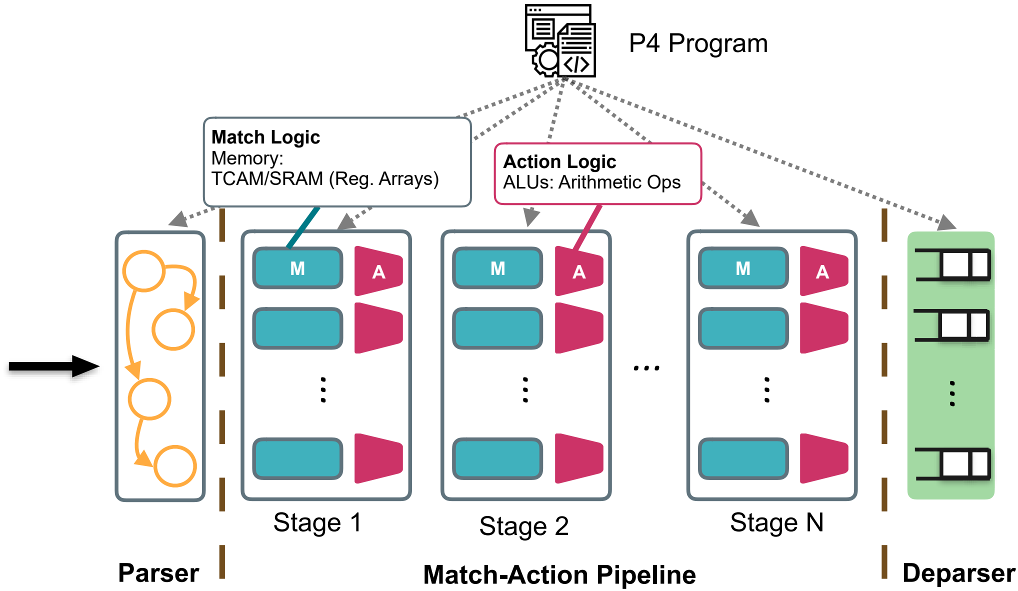


Figure 2.3: Example of a programmable switch architecture.

2.2 Related Work

We summarize the related works in the following.

Software-based Schedulers. There is a long line of research regarding scheduling in distributed systems. Traditionally, schedulers are designed as software processes that run on one or multiple servers. Centralized schedulers focus on service times in the range of seconds to hours and are able to make more complex decisions and support a wide range of resource allocation policies [72, 33, 56]. Multiple prior works, e.g., YARN [71] and Mesos[35], decouple resource allocation from the task scheduling logic. Our work can be complementary to such works as it focuses on realizing the fine-grained task scheduling on the allocated resources in shared clusters.

To scale up and support higher throughput, distributed schedulers have been proposed where they trade off the accuracy for better throughput [55, 28, 23, 24, 40]. The shortest task service time supported by the distributed schedulers is still in the order of hundred milliseconds and such schedulers require large amount of processing resources on the hosts. For example, Sparrow [55] is a distributed scheduler designed for high scheduling throughput. It needs one scheduler node for handling decisions of every 10 worker nodes and still takes between 1 ms to 10 ms for scheduling each task. This is because the schedulers rely on probing *after receiving a task* to get information about the worker states. In such design, it takes two network round trip times (RTTs) when a worker becomes idle before it starts executing the next waiting task. For latency-sensitive applications, the service time can be smaller than a single RTT [12]. That is, for short tasks, the workers will spend most of their time being idle and requesting a task from scheduler rather than executing the tasks.

Saqr is designed for a different operational point and supports tasks in the order of tens of μs and scales to billions of scheduling decisions per second. In contrast to software-based distributed

schedulers, Saqr does not queue the incoming tasks and instead distributes the worker state information to schedule tasks in real-time.

In-network Computing. Emerging programmable data planes enable offloading of various applications to the network to achieve higher throughputs and lower latencies [37, 75, 63, 46]. Most prior works, however, focus on rack-scale designs where a single switch handles all of the traffic for a certain application. Similarly, recent works [76, 43, 42] focused on centralized rack-scale schedulers inside the network. For example, R2P2 [43] and Falcon [42] aim to realize a centralized task queue inside switches to store tasks until a worker becomes idle. R2P2 recirculates a task packet until a worker becomes available, while Falcon stores the task data in a queue data structure in switch memory. A fundamental problem of these approaches is the limited scalability for large clusters: in case of request bursts, R2P2 results in significant bandwidth overheads, and Falcon fails to store the tasks inside the limited switch memory.

The closest work to Saqr is Racksched [76], which uses randomized load balancing for scheduling tasks for a rack of servers. Unlike Racksched, Saqr scales to the datacenter network. We compare Saqr against Racksched.

Load Balancing Theory. There is a substantial body of work in the theory literature regarding the performance of different policies such as power-of-two choices [52, 19] and Join Idle Queue [49, 69]. The power-of-two choices policy has been used extensively in prior software-based schedulers [55, 24]. Our work, however, addresses the specific challenges of realizing a scheduling policy *inside the network* and in *distributed* scheduler setup. We design a new policy that can be efficiently realized inside the datacenter network using a hierarchical architecture. The proposed policy and the state distribution mechanisms in Saqr minimize the communication overheads which were not studied in such setup in the prior works.

2.3 Challenges of In-network Scheduling

In-network scheduling can offer short latency and high throughput for datacenter applications. However, it faces multiple challenges that stem from the design of the programmable switches themselves. We briefly describe the main challenges in the following.

Current programmable switches offer a restricted programming model that operates within a stringent hardware environment. We summarize the main properties of programmable switches that make realizing load-aware scheduling in data plane challenging:

- ***P1: Single access to each register array.*** Each packet in the pipeline can only access one index of the register array and perform a primitive operation on it.

- ***P2: Stage-local register arrays.*** Each register array is bounded to a specific stage at compile time and the memory block is only present to that specific stage.

- ***P3: Limited number of stages.*** The number of stages in the pipeline is limited (e.g., 10–20 stages) which limits the number of allowed sequential operations.

- ***P4: Limited memory resources.*** The total memory available in the switch is limited, and it needs to be shared among applications offloaded to the switch for in-network processing as well as the primary networking functions, with a higher priority for the latter [59].

We note that most of these limitations are based on design choices to enable line-rate processing of packets and ensure atomic changes to registers in the switch pipeline.

P1 makes it challenging to realize even simple sampling-based scheduling policies, such as power-of-two choices, since the switch needs to read two samples from the memory for an arriving task. Previous works, e.g., [76], used one register per server and allocated the registers on different stages to overcome the single-access limitation. This, however, cannot scale to thousands of nodes in the datacenter because of the limited number of stages and registers (P3).

Another challenge that arises from the P1 and P2 is that when schedulers read load values and select a worker node, it is not possible to update the switch view about the worker in a single pipeline pass. Using packet resubmission for every single task introduces a significant amount of throughput overhead. A workaround for this is to offload the load monitoring to the worker nodes and rely solely on the response packets for updating the switch state. This approach, however, results in increased delay of at least 1 RTT, which is sometimes even larger than the task execution time itself for latency-sensitive applications. This also means that a scheduler may make decisions using stale information until one of the tasks finishes executing in the worker and a reply packet updates the switch view. Drift of scheduler view from the actual worker loads can degrade the quality of the scheduling even when using randomized load-balancing mechanisms [22]. To overcome this challenge, Saqr implements a new data structure especially designed for programmable network switches which enables the switches to update their state in real-time after selecting a worker with minimal throughput overheads.

P4 adds another complexity for distributed in-network scheduling. Specifically, in software-based distributed schedulers [55, 24, 23], a common practice is that schedulers probe and acquires the worker state *after* receiving a scheduling request. In contrast, for in-network scheduling, the schedulers are switches with limited packet buffer and on-chip memory. Therefore, it is not feasible to store the tasks in switches, while sending probes and waiting for load information as this would not scale for systems with high task arrival rates. As an example, for a workload with task mean execution time of 100 μs, a cluster of 20K workers can process 200M requests per second. Given a median packet size of 500B and a median RTT in the datacenter of 300 μs [34], a scheduler running on a switch would need at least 30MB of memory for storing the packets while it waits for the probe response, which can be larger than the total buffer available in modern programmable switches.

Chapter 3

Proposed in-Network Scheduling System

This chapter starts with an overview of Saqr and its design principles. Then, it presents the design details for various system components.

3.1 Overview and Design Principles

Overview. Saqr is an in-network, hierarchical, scheduler designed for datacenters that use multi-rooted Clos topologies. We consider the widely-deployed leaf-spine topology as a representative one. Our principles and ideas, however, still apply to other tree-based topologies. In the leaf-spine topology, the top layer consists of core switches that connect different leaf-spine planes. Spine switches provide connectivity between leaf and core layers. Every leaf switch connects a rack of servers to the datacenter network. A server typically hosts multiple virtual machines (VMs) or containers to run tenant applications. An application is composed of *tasks*, which are executed by *workers*. A worker is unit of processing resources, which can have one or more CPU cores. A task is the smallest unit of work to be done by a worker. A long-running process already runs on workers to execute incoming tasks [55, 31]. Therefore, a task does not require launching a new process. Instead, each task describes the operations to be executed and contains the input data if needed. An idle worker starts processing the incoming task immediately. If the worker is busy with another task, the incoming task will be queued at the worker until it becomes idle. Datacenters run multiple subsystems to control and manage the execution of tasks. One subsystem is the resource manager that allocates workers to applications. Each application requires a different number of workers to achieve its objectives such as performance and/or availability. We refer to the set of allocated workers for an application as a *virtual cluster*.

As shown in Figure 3.1, Saqr is composed of leaf and spine schedulers, software monitoring agents, and a centralized controller. The schedulers in Saqr run as data plane programs inside network switches. The schedulers jointly solve two sub-problems: (i) scheduling new tasks (§3.2), and (ii) distributing the current load of workers among schedulers (§3.3). For the first problem, when the scheduler receives a new task, it examines its memory to identify the load of workers and chooses the worker that minimizes the task waiting time. We present a scheduling policy that does not require

queuing incoming tasks inside switches, adapts to dynamic loads, and runs for the whole datacenter at line rate. For the second problem, we present a method for the schedulers to update the state at switches, which balances between the *freshness* of maintained load values and the *communication overhead*.

Saqr agents are lightweight processes that run on servers to monitor the load of workers. They also run a health check mechanism with the leaf controller to enable Saqr to detect and react to worker/server failures efficiently. The agents, however, are not involved in the actual scheduling decisions. Agents attach Saqr headers to packets. Saqr uses a layer-4 header attached, which makes it compatible with and deployable on top of various routing protocols. Figure 3.2 shows the header format for Saqr packets. The *srcID* and *dstID* fields are used to distinguish the components in our system. Since programmable switches only allow index-based access to the register arrays, we assign fixed IDs for components of each layer (e.g., spine switches get 0–N indices). This allows us to use the IDs as a hash for accessing the register arrays corresponding to the components inside the data plane without using complex hashing mechanisms in switches. To forward Saqr packets, each ID is mapped to a specific component. Thus, a switch only needs match-action tables at the last stage to write the destination IP address.

The centralized controller handles system configurations such as application initialization, termination, worker migrations, and switch failures. In case of switch failures, the controller propagates the received failure events to affected switches and clients to modify their internal memory, as detailed in §3.4. The controller also interacts with existing subsystems such as the resource allocation manager to retrieve the placement information for the workers of each virtual cluster. In §3.5, we describe how the centralized controller uses this placement information to select the leaf and spine switches to schedule tasks for every virtual cluster.

At a high level, the task scheduling in Saqr is done as follows: Clients send their tasks to a randomly-selected spine scheduler, which selects a rack for the incoming task and sends it to the leaf scheduler. In each rack, the leaf scheduler schedules the task to one of the workers running in servers physically connected to the switch. After task execution is done, the agent attaches the latest load information of workers to the headers of reply packets. Upon receiving a reply packet, the leaf scheduler updates the load state in its memory and sends an update message to a spine scheduler if needed.

Design Principles. We summarize the principles that guided our design of Saqr.

- ***Load-aware Scheduling Policy***. The load on workers in datacenters is subject to spatial and temporal variations due to resource allocation policies, application requirements, and seasonality nature of workloads. We propose a scheduling policy that does not queue tasks at schedulers, efficiently tracks the idleness and average load of workers and minimizes the task tail response time.

- **Hierarchical Scheduling** The state at switches increases with the numbers of workers and virtual clusters. However, switches cannot maintain this increasing amount of information

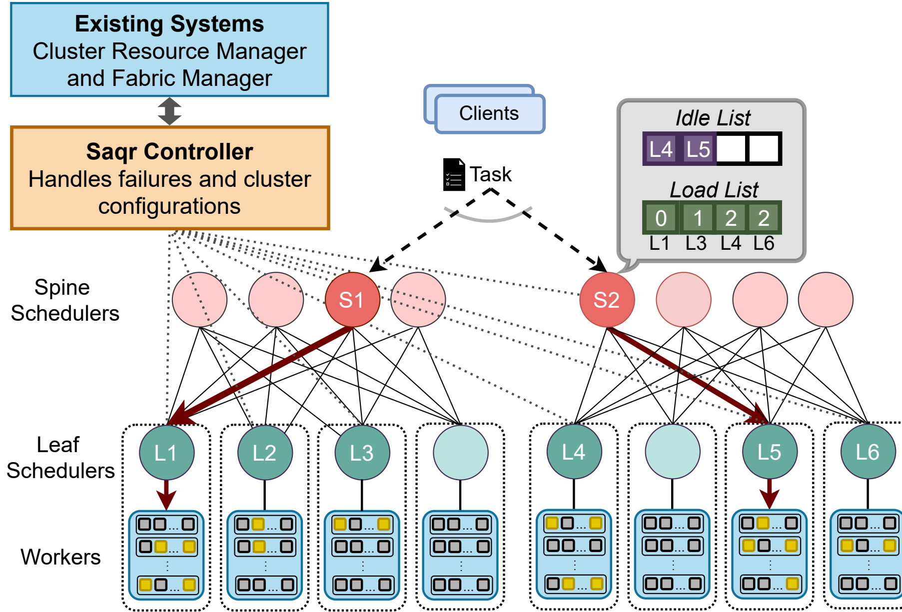


Figure 3.1: Overview of Saqr: The first distributed in-network task scheduler for datacenters.

in their stateful on-chip memory. We carefully divide the load information among switches, and hierarchically manage the collection and update of this information. This significantly reduces the memory and communication overheads.

- **Avoiding State Consistency Overheads** Saqr avoids the complexity of replicating state across switches, by maintaining the load information of a worker or rack at one scheduler. In addition to reducing processing on switches, this principle enables each scheduler to maintain and update its view of a subset of workers without querying other schedulers. Thus, Saqr can achieve a high scheduling throughput with minimum overheads. Saqr also employs a simple failover mechanism to ensure the scheduler availability during failures.

- **Selective State Update** Saqr makes scheduling decisions based on the maintained state without queuing tasks. Thus, updating the state is important to reflect the latest changes. This, however, may increase the communication overhead, despite the hierarchical structure of Saqr. Our idea is that an update does not need to be propagated to a switch *if* that switch can make a decision using its current state. Our approach identifies when an update needs to be propagated by calculating a *drift* between actual load values and the latest load information available at schedulers, and only updates the state of a scheduler when the drift may negatively impact the scheduling quality.

3.2 Scheduling Tasks in the Network

Saqr has two layers of hierarchy consisting of spine and leaf schedulers. Each leaf scheduler tracks the load of individual workers belonging to the same rack, whereas a spine scheduler monitors the load of multiple racks of workers.

3.2.1 Policy Overview

Spine and leaf schedulers employ a similar scheduling policy to assign an arriving task to a lower-layer node. A node for a spine scheduler refers to a rack, whereas it refers to a worker for a leaf scheduler. At a high level, a scheduler maintains two types of state about the lower-layer nodes: (1) IDs of idle nodes and (2) queue length values of nodes. To update their states, schedulers exchange different types of messages, such as *idleAdd* and *idleRemove*, which will be described in the next section.

When a new scheduling request arrives, if the scheduler is aware of an idle node, it will send the task to that idle node. Otherwise, the scheduler takes d samples (assuming $d \geq 2$) from the queue length state of nodes and selects the least loaded node among the sampled values. In our system, we set d to two to balance between the scheduling quality and required memory and processing resources on switches. In literature, this is referred to as power-of-two choices [62].

For spine schedulers, an idle node is a rack that has at least one idle worker because if a leaf is selected, the leaf scheduler will select the idle worker which results in zero queuing time. In addition, we calculate the mean queue length of all worker queues inside a rack as an indicator of how loaded this rack is. We employ the mean value because leaf schedulers balance the load between their tracked workers. We describe the memory layout to represent virtual clusters. Then, introduce our ideas to efficiently realize the proposed scheduling policy for idle and busy nodes.

Memory Layout and Allocation. We maintain load state of nodes in register arrays, which are stateful on-chip memory in programmable switches. Saqr uses separate data structures for storing state of idle nodes (referred to as *idle list*) and storing the queue length of busy nodes. To support virtual clusters, a switch needs to store the state about each virtual cluster in a separate register array. Since the number of register arrays is limited by the number of stages in the switch pipeline, it is infeasible to allocate a single array for each virtual cluster. Instead, Saqr pre-allocates large arrays to store the state for all virtual clusters, and uses the *clusterID* field in packet headers to map each virtual cluster to a base index in the arrays. For simplicity and brevity, we do not show this step when discussing accesses to register arrays, and present the data structure of each virtual cluster as a dedicated one.

3.2.2 Scheduling Tasks to Idle Nodes

We use register arrays to keep a list of IDs of idle nodes, where each slot represents an ID of a single idle node. Since programmable switches only allow one access to the register array per packet, the

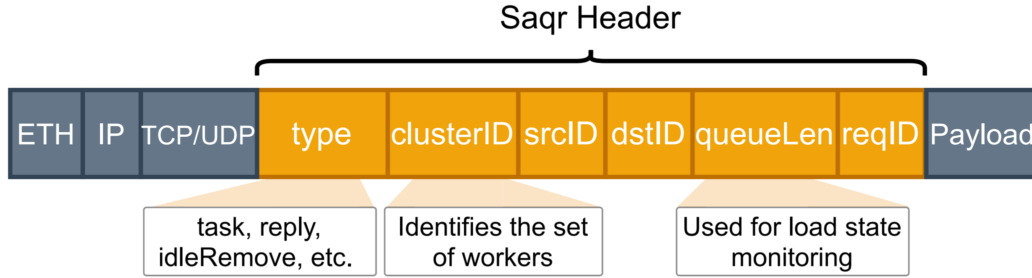


Figure 3.2: Saqr header format.

proposed data structure should support *efficient* GET, ADD, and REMOVE primitives with a single *read-modify-write* operation.[1]

Our insight is that a scheduler needs only to know whether there exists an idle node and retrieves its ID. It does not need to identify the temporal order for when these nodes became idle. Thus, we design a list that guarantees the first contiguous slots to be valid, and the remaining slots to be invalid. This way, a scheduler can always access the top of the *idle list*, while knowing for sure that the top slot is valid. To realize efficient accesses, we use an additional register called p as a pointer to the top of the *idle list*, and a register array called *idleIndex* (maintained only at spine scheduler) to store the index of each idle node. Algorithm 1 lists the pseudo code for handling idle nodes in data plane.

Adding an idle node to the list happens when a scheduler receives a state update packet, i.e., *idleAdd*, about a node becoming idle. The scheduler accesses p and increments its value, then writes the ID of the idle node to the corresponding slot based on the initial value of pointer (Line 2). The scheduler also maintains the current index p in the *idleIndex* register array to be used later. When a new scheduling request arrives, the switch reads p and then retrieves the ID of idle node from the array (Line 9).

Unlike adding an idle node, removing an idle node after being selected by SCHEDULETASK-IDLE depends on the scheduler type. A leaf scheduler needs to remove the selected idle worker immediately after sending a task to it because the worker is no longer idle. For a spine scheduler, however, sending a task to an idle leaf does not necessarily mean that the rack is not idle anymore. This is because the rack may have more than one idle worker. Therefore, a spine scheduler only reads p and does not decrement it. Instead, removing an idle node in a spine scheduler is triggered by an *idleRemove* packet sent by a leaf scheduler.

Upon receiving an *idleRemove* packet, the spine scheduler examines the *idleIndex* array to check the index of $pkt.srcID$ in the *idle list*. The spine scheduler resubmits the packet with the current last node ID $lastNodeId$ and the index of the node to be removed $removedNodeIdx$. In the resubmission path, the scheduler only places the $lastNodeId$ in the slot of $removedNodeIdx$ and updates the $idleIndex$ accordingly. With this approach, instead of removing items from the array,

[1] An atomic operation that prevents race conditions in the pipeline.

Algorithm 1 Scheduling and state updates for idle nodes

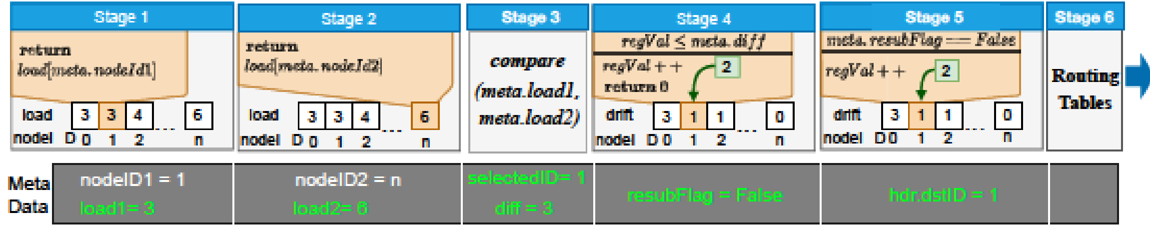
- p: reg. pointing to the next available slot in idleList
- idleList: reg. array that holds the IDs of idle nodes
- idleIndex: reg. array holding indices of the nodes in idleList.

```
1:  // On an idleAdd pkt received
2:  function ADD(pkt)
3:      readInc(p)
4:      idleList[p] ← pkt.srcID // pkt.srcID is idle node ID
5:      idleIndex[pkt.srcID] ← p
6:  function GET()
7:      readDec(p) // Only a leaf scheduler decrements p
8:      return idleList[p]
9:  function SCHEDULETASKIDLE(pkt)
10:     selectedNode ← GET()
11:     Update pkt with IP of selectedNode and Forward
12: (A) On an idleRemove pkt received: First Path
13: function REMOVE(pkt)
14:     removedNodeIdx ← idleIndex[pkt.srcID]
15:     readDec(p)
16:     lastNodeID ← idleList[p]
17:     resubmit(lastNodeID, removedNodeIdx)
18: (B) On an idleRemove pkt received: Resubmit Path
19: function REMOVE(lastNodeID, removedNodeIdx)
20:     idleList[removedNodeIdx] ← lastNodeID
21:     idleIndex[lastNodeID] ← removedNodeIdx
```

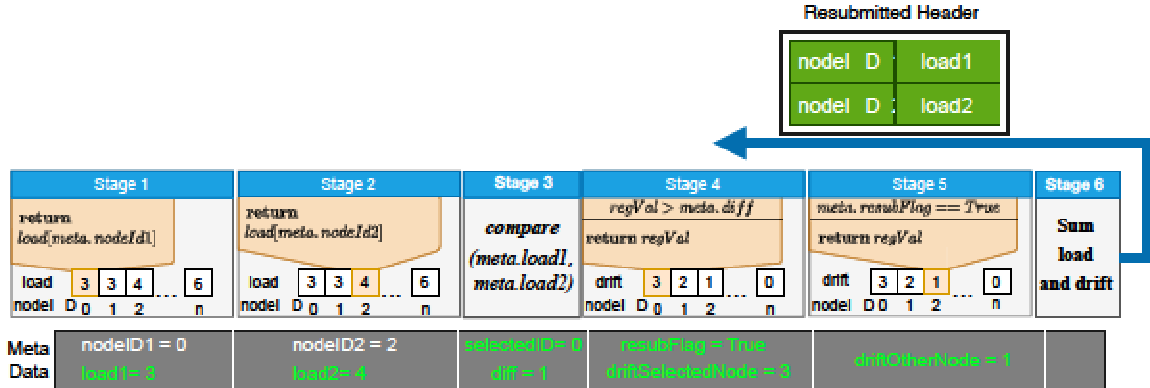
we keep reusing the memory locations and validate/invalidate them based on the pointer value. We note that current programmable switches does not preserve the order of processing of resubmitted packets. For example, before processing the resubmission path of an $idleRemove$ packet A, the first path of another $idleRemove$ packet B could be processed by the switch. In such case, the index of $removedNodeIdx$ observed by B might be incorrect since the linearization point of procedure is done after processing the resubmission path. To handle this issue, we use a single register as a logical lock for the REMOVE operation, where the lock is acquired in the first path and released in the resubmission path. In case of unsuccessful removal, the spine scheduler drops the packet, and the leaf scheduler will resend another *idleRemove* packet after receiving another task from the linked spine.

3.2.3 Scheduling Tasks to Busy Nodes

When there is no idle node, the scheduler examines a list of load values of the tracked nodes, which is called the *load list*. Unlike scheduling tasks to idle nodes, realizing the power-of-two choices for busy workers is more challenging. This is because a scheduler needs to read two randomly selected indices from the *load list* and select the least loaded one. The first challenge is that it is not allowed



(a) Case 1: Schedule a task using a stale state and update the *drift list* in one pass



(b) Case 2: Resubmit a task packet to update the state, and schedule the task based on a fresh state

Figure 3.3: Illustrative example of scheduling tasks to busy workers with selective state updates in Saqr.

to read more than one item per packet from the same register array. Second, the scheduler needs to scale to thousands of nodes and achieve fast access. Finally, after the scheduler makes a decision, it should keep a fresh view of the load values while reducing the processing overhead.

To address the first two challenges, Saqr maintains two identical copies of the *load list* in two different stages, where each array stores the load of all downstream nodes (one node per slot). Storing two copies allows the scheduler to read one random index from each copy and then compare them. When a new state update for a node arrives at a scheduler, it writes the updated load value on both copies. Prior works, e.g., [76], can maintain only load values in the order of number of pipeline stages (i.e., 10–20 nodes). This is because it allocates one register for every load value and places them at different stages to allow reading multiple values in a single pipeline pass.

For addressing the third challenge, a scheduler should update its local view on the load information (i.e., the *load list*) *after making scheduling decisions.* This requires the scheduler to write back the updated load value to the corresponding register slot. As described earlier, the same register array cannot be accessed twice per packet. A straightforward solution is to resubmit each packet to the pipeline and update the load state on the second pass. This, however, results in a significant processing overhead, increases the scheduling latency, and doubles the bandwidth required for task packets. We note that conventional *double buffering* techniques can not be used to update the scheduler view of busy workers. In that case, one register array will be used for reading values and another one will be used for writing the modified values. However, double buffering can not be realized in the data

plane since it violates the P2 property, mentioned in §2.3. As an example, consider buffer A to be placed at stage x and B at a later stage $x + 1$. For the first packet, switch can read from A, select the node and write back the incremented value to B. For the next packet, first it needs to read B and then write back to A which contradicts the initial register stage placement.

Our key idea is that a scheduler keeps scheduling tasks using the stale view of the *load list* until it detects an update is needed. The scheduler resubmits the packet to the pipeline only when the *load list* needs to be updated. We refer to this idea as *selective state update*. In our solution, the scheduler maintains a *drift* value per node to measure how far the maintained load of a node is from the actual load of the same node.[2] Specifically, the scheduler maintains two copies of the *drift list* placed in two stages. Figure 3.3 illustrates an example of the proposed solution.

When a scheduling request arrives, the scheduler reads two samples from the copies of *load list* and calculates the difference between the load values. Next, the scheduler accesses the first copy of *drift list* to check how many more tasks are actually queued in the node that the scheduler initially selected as least loaded. If the drift value is lower than the difference between load values (Figure 3.3a), the scheduler increments the corresponding drift value in each copy of the *drift list*. Otherwise, the initial scheduling decision may be underestimating the actual load of the node (Figure 3.3b). Thus, the scheduler reads the drift value for the second node from the second copy, calculates the actual load of a node as the sum of drift and load values, and resubmits the packet with this data. In the resubmission path, the scheduler becomes aware of the actual load values and it selects the least loaded node. The *load list* is updated with the new load value, and the *drift list* is reset indicating that the load value in the *load list* is up-to-date. Notice that in the resubmission path, the scheduler does not read items from the *load list* or *drift list* as these values are injected in the resubmitted packet. Thus, the scheduler guarantees to update the *load list* and *drift list*.

Number of Samples in Power-of-d Choices. The power-of-d choices algorithm has been analyzed in the literature using the mean field theory [52]. We summarize this analysis here and relate it to our datacenter scheduling problem.

Consider n workers serving tasks in first-come-first-serve (FCFS) manner. Tasks arrive according to a Poisson process with rate $n\lambda$. The task service time is assumed to have an exponential distribution with mean 1. Upon receiving a task, the scheduler samples d workers independently and uniformly at random from the n workers and chooses the worker that currently has the fewest tasks in its queue. The task will then wait inside the queue of the selected worker until the worker becomes available to serve the task. Within this setup, the total time that a task spends in the system (i.e., response time) consists of the task waiting time in the worker queue and the task service time. The analysis provides bounds on the expected task response time based on the number of samples.

It has been shown that setting $d = 2$ yields an exponential improvement in the expected task response time compared to randomized worker selection (i.e., $d = 1$). Increasing d to greater than

[2]Since the load value of every node is maintained at only one scheduler, schedulers do not face issues such as race conditions.

two, however, decreases the maximum task response time by just a constant factor. Particularly, the maximum queue length of workers over a period T will be $\frac{lnln(n)}{ln(d)} + O(1)$ with *high probability* p, where $O(1)$ depends on λ (i.e., load) and the duration T, and the probability $p = 1 - O(1/n)$. That is, for sufficiently large n, the expected response time improves exponentially when $d = 2$ compared to the case of $d = 1$. Furthermore, their simulations show that even for relatively small n (e.g., 100 workers), the system is expected to follow the same behaviour.

We note that, sampling more load values (increasing d) can be realized inside the switch, but it requires significant amount of memory and processing resources on the switches. Specifically, to find the smallest value among d samples, we need at least $log(d)$ sequential operations, which also means we need at least $log(d)$ stages of the switch processing pipeline for this process (refer to the P3 property mentioned in §2.3). Also, reading d samples from the memory requires access to d different memory locations while based on the P1 property (§2.3), each register array can be accessed only once per packet. In addition, the scheduler needs to update the load value for the selected worker in its memory while the P1 property does not allow writing back the incremented value on the same register array. As discussed in §3.2.3, we use additional data structures and a new algorithm to overcome the mentioned challenges for realizing power-of-two choices inside the switch while keeping a fresh state after each decision. We evaluate the impact of number of samples on the performance of Saqr in §6.3.

3.3 Distributing State Among Schedulers

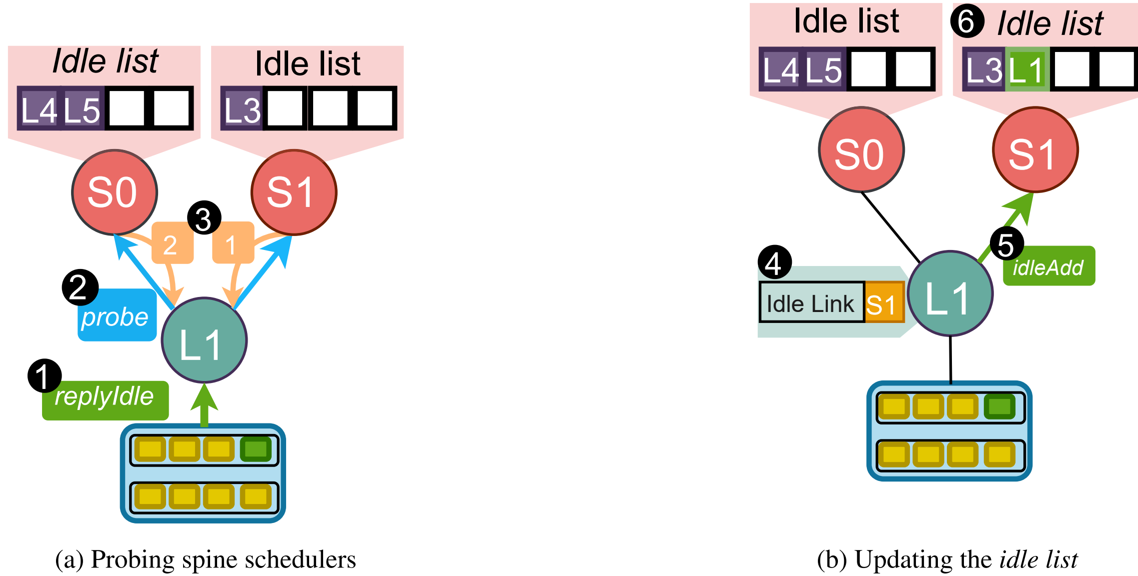
We design simple mechanisms to distribute the necessary information among leaf and spine schedulers to update their states. This enables the execution of the proposed scheduling policy using fresh information with minimal overheads.

3.3.1 Distributing Worker State to the Leaf Layer

Since scheduling tasks to workers of a rack is only done by leaf schedulers, each leaf scheduler updates its state when selecting a worker for a task as described in §3.2. When a task is done executing on a worker, the agent modifies the *queueLen* field in the header (Figure 3.2) and uses the reply packets to report the updated load to the leaf scheduler. If the reply packet indicates that the worker is idle, the leaf scheduler adds the *srcId* to the *idle list* (Algorithm 1) and updates the *load list* for the corresponding index.

3.3.2 Distributing Rack State to the Spine Layer

We consider two types of information to be distributed to the spine layer: (i) idleness of the rack, and (ii) average load of the rack. Instead of naively sending every change to any spine scheduler, we define two *state linkage* primitives to send information about each rack to only one spine scheduler.
Idle Linkage. When a leaf scheduler becomes aware of an idle worker, it sends probe packets to two randomly-selected spine schedulers and asks for the number of idle racks each spine scheduler



(a) Probing spine schedulers

(b) Updating the *idle list*

Figure 3.4: Distributing idle state from a leaf to spine scheduler.

is aware of. Then, it sends an *idleAdd* packet to the spine scheduler that has the smallest number of idle racks. After this step, the spine scheduler will know that the sending leaf scheduler has idle workers and sends incoming tasks to it. Figure 3.4a shows the probing step of idle linkage process and Figure 3.4b shows the last step in linkage where the leaf selects the spine scheduler to link with. The probing mechanism in our design is similar to the [49] which allows us to distribute the information about idle racks among the spine schedulers. Using this simple mechanism, the probability of the event that a randomly chosen spine is aware of some idle racks is increased and therefore it improves the scheduling quality. We also note that Saqr can be easily configured to avoid using probing for idle linkage and randomly send the *idleAdd* to one of the spine scheduler to reduce the linkage delay and communication overheads. Once there are no more idle workers available in the rack, the leaf scheduler sends an *idleRemove* packet to the linked spine which removes the leaf from its *idle list*.

Load Linkage. Each spine scheduler in our system tracks the average load of a subset of racks. Given that workers are placed on L racks and S spine schedulers are selected, each spine tracks the average load of $R = L/S$ racks. The controller divides the racks in disjoint subsets of size R and assigns each subset to one of the spine schedulers. It also sends the IDs of the leaf schedulers of the selected racks to each assigned spine scheduler. Each leaf scheduler receives and stores the ID of the linked spine scheduler for the virtual cluster, and uses this information for sending load state updates to it later. This way, we divide the state about the load of the racks among the spine schedulers, and ensure that load state of each rack is only tracked by one spine scheduler. Figure 3.5 shows an example of load linkage for a virtual cluster with workers that span four racks ($L = 4$), and uses two spine schedulers ($S = 2$). Using this linkage primitive, Saqr avoids replicating load information among multiple schedulers.

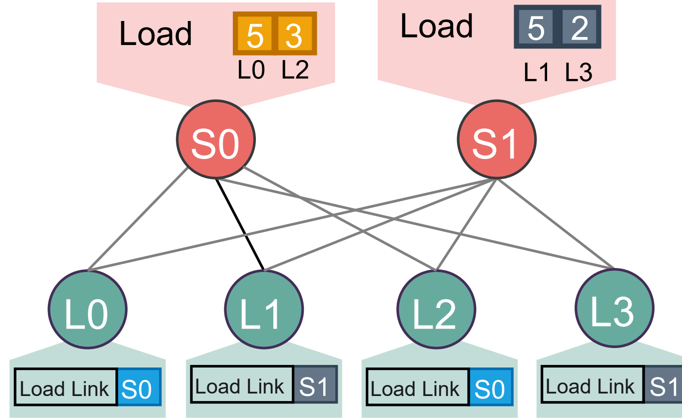


Figure 3.5: Distributing load information from leaf to spine schedulers.

Realizing Average Queue Length. Calculating an average value in the data plane is infeasible due to the lack of support for floating point operations in programmable switches. To handle this, we define a convention between schedulers to use a fixed-point representation for the average using a 16-bit value where 5 bits represent the fraction part and 11 bits show the decimal part. When cluster resources are allocated, the leaf controller calculates the fixed-point representation of $1/\#workers$ and stores this value in a table. In the data plane, the leaf scheduler uses the *clusterID* as a key to access the table, and uses add/subtract to increase/decrease the average value for every change. The 5-bit fixed-point representation has a maximum conversion error of 0.015625, i.e., $2^{-(5+1)}$. Saqr can trade a larger header size for a better precision.

Selective Updates From Leaf to Spine. Instead of sending the updated average load to a spine scheduler after each change, the leaf scheduler only sends a state update packet when the view of the spine scheduler is drifted from the actual average load by a threshold of $T = 1$. To realize this, when a task arrives at a spine scheduler, it assigns the task to a rack and increments the average load of the selected rack in its memory (as described in §3.2). Also, each leaf scheduler passively tracks the average load of the rack by observing the incoming tasks from spine schedulers and reply packets from workers. Therefore, each leaf scheduler detects when the drift of a spine scheduler view is larger than the threshold and sends a state update packet with type *loadSignal* to that spine scheduler which contains the latest average load for the rack. This allows Saqr to reduce the number of update packets sent to spine schedulers by the factor of number of workers in the rack without sacrificing the scheduling performance.

Assuming that load is perfectly balanced between the workers in each rack, this approach results in the same waiting time as tracking the actual real-time average. When a spine scheduler view shows an average load μ for a rack while the actual average is $\mu \pm \Delta$. When $\Delta < 1$, the new task will be assigned to a worker with at most μ pending tasks. On the other hand, when $\Delta \geq 1$, it means that every worker in the rack has ± 1 pending tasks in their queues. Therefore, a spine scheduler needs to be updated.

3.4 Handling Failures

In this section, we introduce the mechanisms for handling failures of different components of Saqr.
Spine Switch Failure. We rely on existing network protocols [73, 53, 41] for detecting failures and we assume that the Saqr controller receives the failure event from the fabric manager.

A failed spine switch can be in charge of scheduling for multiple virtual clusters depending on worker placements and the initial spine selection algorithm. Upon receiving a failure event, the controller first checks the virtual clusters that were using the failed spine as scheduler and does the following to mitigate the impact on applications: (1) Send the failure event to the clients of impacted virtual clusters. Each client removes the failed spine scheduler from the list of schedulers to avoid sending future tasks. (2) Send the ID of failed spine to the leaf switches that were in the scheduling path of the impacted virtual clusters. The control plane of leaf switch needs to modify the number of available spine schedulers and reset the state linkage registers if the leaf scheduler was linked to the failed spine scheduler. This ensures that leaf schedulers do not send the state updates to the failed spine scheduler anymore and report to the other available spine schedulers.
Leaf Switch Failure. Similar to a spine failure, the controller sends the ID of the failed leaf to all spine schedulers of impacted virtual clusters. Each spine controller removes the failed leaf from the idle and load arrays (Line 13 in Algorithm 1) to avoid sending tasks to the failed rack. When the leaf switch recovers from the failure, the Saqr controller sends the information about the spine schedulers and workers in the rack to the leaf switch. The leaf controller then sends a *hello* message to all servers in the rack, and the worker agents reply with their load. The leaf scheduler distributes the state using the method mentioned in §3.3.
Worker Failures. To detect worker failures, the leaf controller exchanges heartbeat packets at a fixed rate with the server agents. The leaf controller locally detects the failures of the workers in the rack based on a fixed timeout value. Since Saqr handles worker monitoring at rack-level, detecting and handling failures can be done independently in each rack. After detecting a failure, the leaf controller decreases the number of available workers and instructs the data plane to remove the failed worker from its data structures.
Controller Failure. We assume that the logically centralized controller is replicated on multiple servers using existing practices, e.g., Paxos [44], and it is reliable. We note that the Saqr controller is only essential for virtual cluster re-configurations and handling switch failures while scheduling and state update operations are done entirely inside the data plane.

3.5 Selecting Spine Schedulers

At leaf layer, all of the switches that are physically connected to the workers are in charge for monitoring workers and scheduling tasks on them. For spine schedulers, the controller selects S switches upon receiving the initial configuration request from the cluster resource manager. The spine switches that handle the task will be selected from the switches in the pods that workers are

located to avoid sending Saqr-related traffic to other pods that do not have any workers. We select more than one spine scheduler to distribute the traffic load on the switches as one switch might not be able to handle the aggregate throughput for M rack of workers. In addition, the distributed approach ensures high availability in case of spine switch failures.

Applications can request the minimum number of required spine switches S_{min} depending on the availability requirements. We assume that network operator allows a fraction of switching capacity of each spine switch to be consumed for handling scheduling-related packets. Therefore the controller gets the available capacity in terms of packet per second for each spine switch $= [C_i, C_2, ..., C_k]$ as an input. Based on the number of allocated workers in each virtual cluster N, and the minimum processing time of task by workers T_{min}, we provision the maximum task arrival rate that needs to be handled by the schedulers as $R_{max} = \frac{N}{T_{min}}$. Beside the processing requests, switch needs to process the reply packets for state updates from the workers once they finish executing the tasks (§3.3) which will be at worst case equal to R_{max}.

For the initial configuration request from the resource manager, the controller run a best-effort greedy algorithm to select the spine schedulers based on the given inputs while satisfying the capacity requirements of each switch. If the packet rate is beyond the remaining capacity of the spine switches in the pod, the algorithm divides the load on more switches until it can satisfy the requirement. Because of the dynamic nature of virtual clusters it is not valid to assume all of them are initialized at time zero and therefore the proposed solution will be sub-optimal in terms of using switch capacity. If the algorithm fails to find switches with required capacity even with distributing the load upon all of the pod switches we return an error meaning that no more clusters can be placed for the selected resources.

Chapter 4

Implementation and Practical Considerations

4.1 Implementation of Saqr

We have implemented a proof-of-concept of Saqr consisting of leaf and spine schedulers and monitoring agents. Both scheduler programs are implemented in P4 [16] based on the algorithms described in Chapter 3.

Current programmable switches have limited flexibility for generating new packets inside the data plane. Packet generation can only be triggered using periodic timers or on port failure events. In Saqr, the leaf scheduler might need to send a new update message to spine and this event is triggered upon *receiving a packet*. To realize the communication between Saqr schedulers for state propagation, we use the *mirror* (also known as *clone*) feature in programmable switches to make a copy of the original packet. Upon receiving a packet, if a leaf scheduler needs to send an additional state update to spine schedulers, it makes a copy and sends the copy to the destination based on the original header fields. Then, it changes the original packet headers accordingly and sends this version to the spine scheduler. For example, consider the case when a leaf scheduler receives a reply from a worker and it needs to send a *loadSignal* packet to update the load list of a spine scheduler. In this case, the leaf scheduler will mirror the reply packet to its destination address so that the results are forwarded to the client. Then, it modifies the same packet by setting the *type* field to *loadSignal* and the *queueLen* field to the latest average load of the rack. It then sends the packet to the spine scheduler.

At servers, we implemented a modified version of Shinjuku scheduler [39] to run inside each machine and added our monitoring logic on top of it. The OS scheduler inside each server runs Dune [14] for process virtualization. Unlike Shinjuku, Saqr does not use a shared queue for all of the worker cores, and it uses separate per-worker queues to isolate resources in the multi-tenant environments. Saqr schedulers select the worker core(s) based on the virtual cluster, and inside each server we enqueue/dequeue the task to the corresponding worker queue. We implemented the monitoring agent in C on top of Shinjuku with less than 100 lines of code. The monitoring agent

adds the queue state information to the reply packet headers after task execution is done. Similarly, Saqr can be integrated with other existing OS schedulers to run the tasks on CPU cores.

4.2 Practical Considerations

Deployability. Saqr does not dictate a specific routing protocol in the datacenter and it can work with any of the existing ones since it utilizes layer-4 headers. In addition, Saqr does not need to be deployed on all switches in the datacenter. Thus, Saqr can be incrementally deployed in datacenters that have programmable and legacy switches. For example, Saqr leaf schedulers can be deployed only on leaf switches of racks where the workers for latency-sensitive applications are placed. Also, Saqr spine scheduler can be deployed on a subset of the switches. This is because the controller gets the information about the available Saqr schedulers running on programmable switches, and selects the schedulers for the given resources (as described in §3.5).

Handling Multi-packet Tasks. When a task is composed of multiple packets, Saqr schedulers need to send these packets to the same worker. Saqr can leverage prior approaches, e.g., [50, 76], to support multi-packet tasks by maintaining state for multi-packet tasks inside the switches. In particular, schedulers only make a scheduling decision for the first packet of a task, and maintain in memory a mapping between $\langle clusterID,\ reqID \rangle$ and the chosen node. For subsequent packets of the same task, a switch retrieves the previously selected node to forward the packets to. Although this approach consumes additional memory resources at switches, the distributed design of Saqr allows it to handle larger scales compared to centralized schedulers. This is because the total load can be divided between multiple spine schedulers. Also, each leaf scheduler only needs to maintain state for the subset of requests sent to the rack.

In addition to the mentioned stateful approach, a stateless approach can be used to handle multi-packet tasks as proposed in prior works (e.g., [58, 11, 43]). In this case, the switches do not maintain the state of the connection for the packets of the task. Instead, the selected worker sends a reply to the client after it receives the first packet of the task. The client will then send the remainder of packets to the selected worker using normal procedures.

The two approaches make a trade-off between the latency and switch memory resources. Ideally, both stateful and stateless approaches can be employed by Saqr for different workloads in the datacenter. Workloads with low latency objectives can use the switch memory and other workloads can employ the stateless approach.

Handling Heterogeneous Workloads. Today's datacenters host a diverse set of applications including latency-sensitive online services and best-effort batch workloads that run in the background [26, 48, 21]. The tasks for batch workloads can be many orders of magnitude longer than the tasks for online services. Saqr is designed to realize low-latency scheduling for short tasks in the datacenters. In Chapter 6, we used workloads consisting of tasks of up to two orders of magnitude difference in service time to evaluate Saqr, and our results show that Saqr can substantially improve the performance compared to the state-of-the-art. However, adding further diversity in the

workloads could impact the performance of Saqr. Specifically, for extremely diverse workloads, the worker queue length would no longer be a predictor of task waiting time, and short tasks might be blocked behind the long ones in the queues. There are two common practices in the literature for handling heterogeneous workloads in datacenters, and Saqr can support them as follows.

The first approach disallows latency-sensitive services from sharing resources with other workloads, and it deploys such services on dedicated servers [27, 72, 20]. Saqr, by-design supports such mechanism via the virtual cluster abstraction. The dedicated workers for latency-sensitive tasks form a separate virtual cluster of workers and will be handled in isolation by Saqr schedulers. This approach can provide guarantees on tail latencies for short tasks by reserving resources at the cost of possible under-utilization of resources [48].

As an alternate approach, both type of workloads can be collocated on the same workers to improve the utilization by taking advantage of the unused capacity on the machines for running batch workloads. Similar to [55, 76], Saqr can support priority-based scheduling for the shared workers while assigning higher priority to latency-sensitive tasks and using multiple queues to avoid blocking them behind long tasks of batch workloads. In this case, each worker maintains one queue for each priority type and Saqr schedulers keep track of the state of queues for each priority. When a task arrives at the scheduler, it looks up the task priority and based on that checks the state of the corresponding queues for making decisions. When task arrives at the worker, it will be enqueued in the corresponding queue and the worker will fetch the tasks from these queues according to their priorities.

Intra-Server Performance Isolation. To guarantee performance isolation within servers and across workers, we also need mechanisms to detect and prevent interference among workers [54, 48]. This is especially important in multi-tenant environments where VMs of different tenants can be placed on a single physical server [21]. We note that, such mechanisms are orthogonal to our work and are handled by the operating systems running on the end-hosts [60, 39, 30] or VM placement algorithms that handle resource allocation for tenants [70, 36]. Saqr addresses fine-grained task scheduling on the *workers across multiple servers* and it can be integrated with existing intra-server operating systems and schedulers (e.g., Shinjuku [39], which we used in our prototype implementation) that run on these servers.

Limitations and Extensions. Deploying Saqr in graph-based datacenter networks, e.g., Jellyfish [67] and Xpander [68], may introduce multiple challenges. For example, Jellyfish's lack of structure requires new algorithms to divide the load information between more than two layers.

Chapter 5

Evaluation in a Testbed

In this chapter, we evaluate Saqr in a testbed and compare its performance and overheads versus the state-of-the-art in-network scheduler.

5.1 Testbed Setup

Hardware. Our testbed, illustrated in Figure 5.1, has one 3.2 Tbps Intel Tofino switch, which has two hardware pipelines. We configure one of the hardware pipelines as a spine switch and run the spine scheduler of Saqr on it. We emulate four leaf switches on the other hardware pipeline, where each switch represents a rack of servers. We connect the leaf switches to the spine switch using 100 Gbps links. Leaf switches run the leaf schedulers of Saqr. In addition, the testbed has seven servers connected to the leaf switches through 10 Gbps links, each server is equipped with an Intel 82599ES 10 GbE NIC. Up to two of these servers are used as clients to generate scheduling requests and the others are used to host workers to execute these requests.

We consider two settings for the distribution of workers across racks: (1) Uniform: a total of 32 workers are uniformly distributed across all racks, where each rack has eight workers running on a physical server attached to the leaf switch (1 worker/core). The 32 workers run on four identical servers, each has Intel Xeon E-2186G CPU, 3.80 GHz, 12 cores, 32 GB memory. (2) Skewed: a total of 48 workers are distributed as follows: two racks have four workers each, one rack has eight workers, and one rack has 32 workers (running on three physical servers, where one of the servers has has Intel Xeon E5-2650 CPU, 2.3 GHz, 40 cores, 128 GB memory and runs 16 workers and the other two servers have the same specifications as the ones used in the Uniform setup and run 8 workers each).

Systems Compared Against. We compare the performance of Saqr against Racksched [76], which is the state-of-art in-network scheduler designed for short tasks within a single rack. To scale it beyond a single rack, we implemented two variations of Rachsched as follows: (i) *Racksched-Hierarchical (RS-H)*, where instances of the scheduler are replicated to spine and leaf layers (Figure 5.2a), and (ii) *Racksched-Random (RS-R)*, where each rack has its own scheduler, and clients send tasks to one of the leaf schedulers randomly (Figure 5.2b). For RS-H, similar to Saqr, we allow

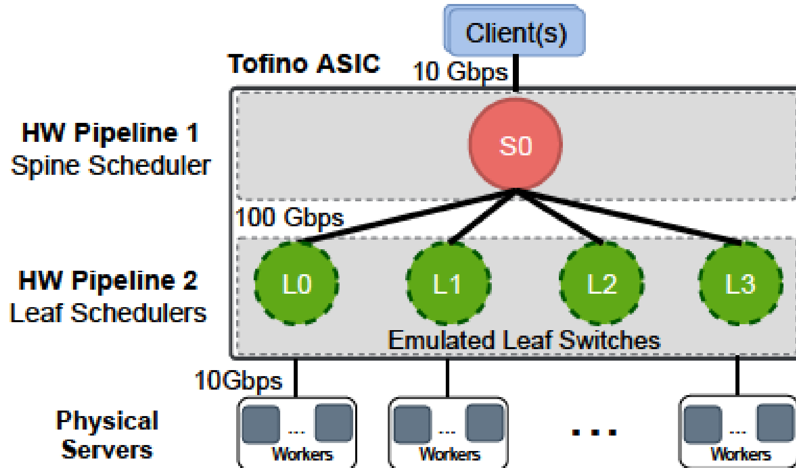
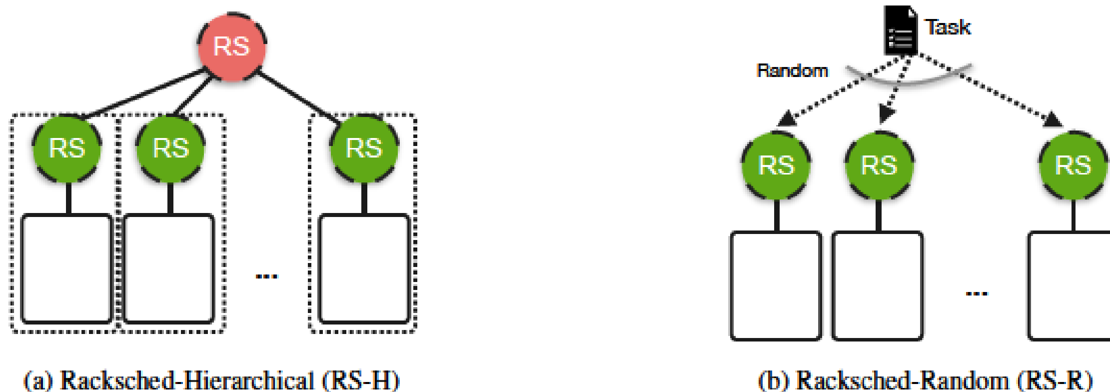


Figure 5.1: Testbed setup. The hardware resources of a Tofino switch are partitioned to emulate one spine switch and four leaf switches. Multiple servers are connected to leaf switches.



(a) Racksched-Hierarchical (RS-H)

(b) Racksched-Random (RS-R)

Figure 5.2: Variations of Racksched.

each spine scheduler to track the average load state of racks and use leaf schedulers to select workers inside each rack. Also, we assume each leaf sends the load state update from workers to every spine scheduler, which allows the spine scheduler in RS-H to have the same view as Saqr over all workers.

We extended the open-source P4 implementation of Racksched [4] to support multiple racks and large number of workers, by implementing the RS-H and RS-R variations. We have validated our implementation using the Packet Testing Framework (PTF) in the software switch model provided by Intel before running our experiments.

Workloads. We implemented a latency-sensitive application that uses RocksDB [8], which is a key-value store used in production by online services such as Facebook. The client generates and sends requests to the network, and the schedulers receive these requests and schedule them to workers for execution. The workload consists of a combination of SCAN and GET requests. A SCAN request scans 5K objects with a median service time of 650 μs. A GET request retrieves 60 objects where

the median time of each request is 40 μs. We use two workload distributions of to evaluate the schedulers under different practical scenarios. The first distribution consists of 50% GET and 50% SCAN requests, whereas the second one. We assume no prior knowledge about the service times and use the same priority queues for both SCAN and GET tasks.

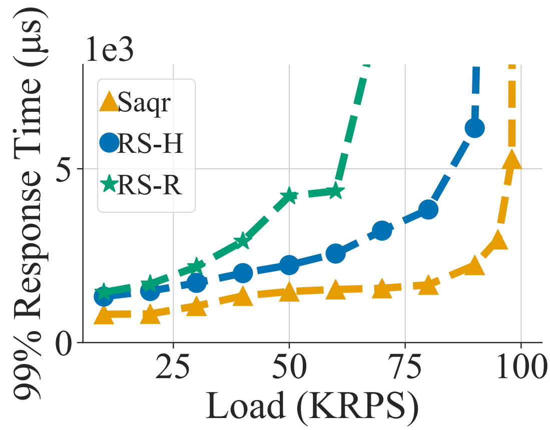
5.2 Experiments and Results

We vary the system load by incrementally increasing the number of requests submitted for scheduling. The system load is measured in kilo requests per second (KRPS) and we keep increasing it until we reach the capacity of the system, where the response time becomes unacceptably high for latency-sensitive applications (e.g., seconds or even minutes for tasks that should complete in micro or milliseconds). The response time is defined as the time from when a task arrives at a spine scheduler until it finishes execution on a worker, and it is the most important metric for scheduling systems. Most datacenter scheduling systems strive to optimize the *tail response time* which is defined as the 99th-percentile response time for the tasks. The tail response time is important because in a real-world setting, an application/job cannot complete until the result of its *last task* is ready.

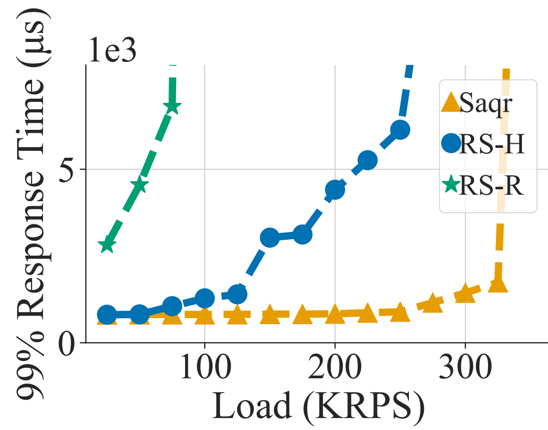
Another important metric for scheduling systems is the achievable throughput, which we define as the maximum system load that can be processed while achieving a given bound on a target performance metric, e.g., the 99th-percentile of the response time should not exceed 3ms.

Response Time. We measure and report the tail response time for different workload distributions and worker setups in Figures 5.3 and 5.4. The figures show that Saqr consistently and substantially outperforms RH-H and RS-R across all workloads and worker setups. For example, in the Uniform worker setup with 90% GET and 10% SCAN requests (Figure 5.3b), Saqr reduces the 99th-percentile of the response time by 85% compared to RS-H when the system load is 250 KRPS; RS-R could not support this load. This also means that Saqr can achieve much higher throughputs (i.e., process higher system loads for any given target response time) than RS-H and RS-R. For the same example in Figure 5.3b, if the target 99th-percentile of the response time is 2.5 ms, Saqr can achieve a throughput of up to 325 KRPS, whereas RS-R and RS-H can only achieve up to 25 and 150 KRPS, respectively. That is, Saqr can improve the throughput by up to 13X and 2.17X compared to RS-R and RS-H, respectively, in this case.

Communication Overheads. We measure the number of update messages per second exchanged between leaf and spine schedulers. The results for different workloads and worker placement setups are shown in Figure 5.5. The figure shows that compared to RS-H, Saqr reduces the rate of state update messages for all load conditions. For example, for the Skewed worker setup and 90% GET and 10% SCAN requests (Figure 5.5d), the rate is reduced by up to 31.5X when the system is lightly loaded (50 KRPS) and up to 5.1X when system is heavily loaded (425 KRPS). We note that when the system load is 425 KRPS, the rate of idle linkage messages in Saqr is reduced because racks have less idle workers and, therefore, the rate is not monotonically increasing as in RS-H.

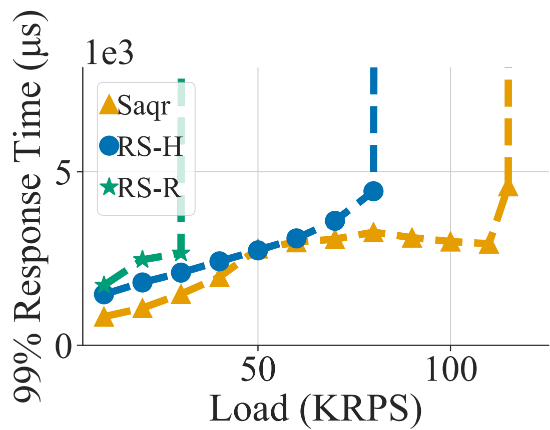


(a) 50%-GET, 50%-SCAN

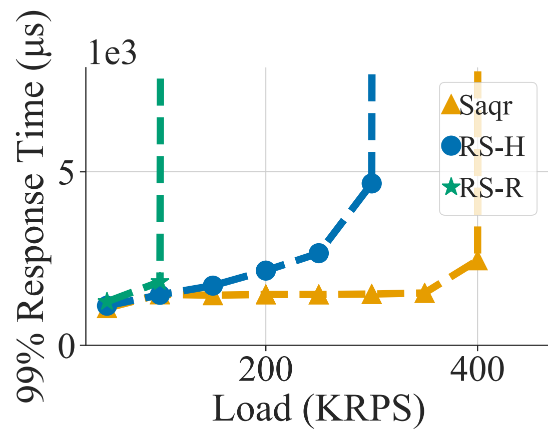


(b) 90%-GET, 10%-SCAN

Figure 5.3: Response time in the Uniform worker placement.

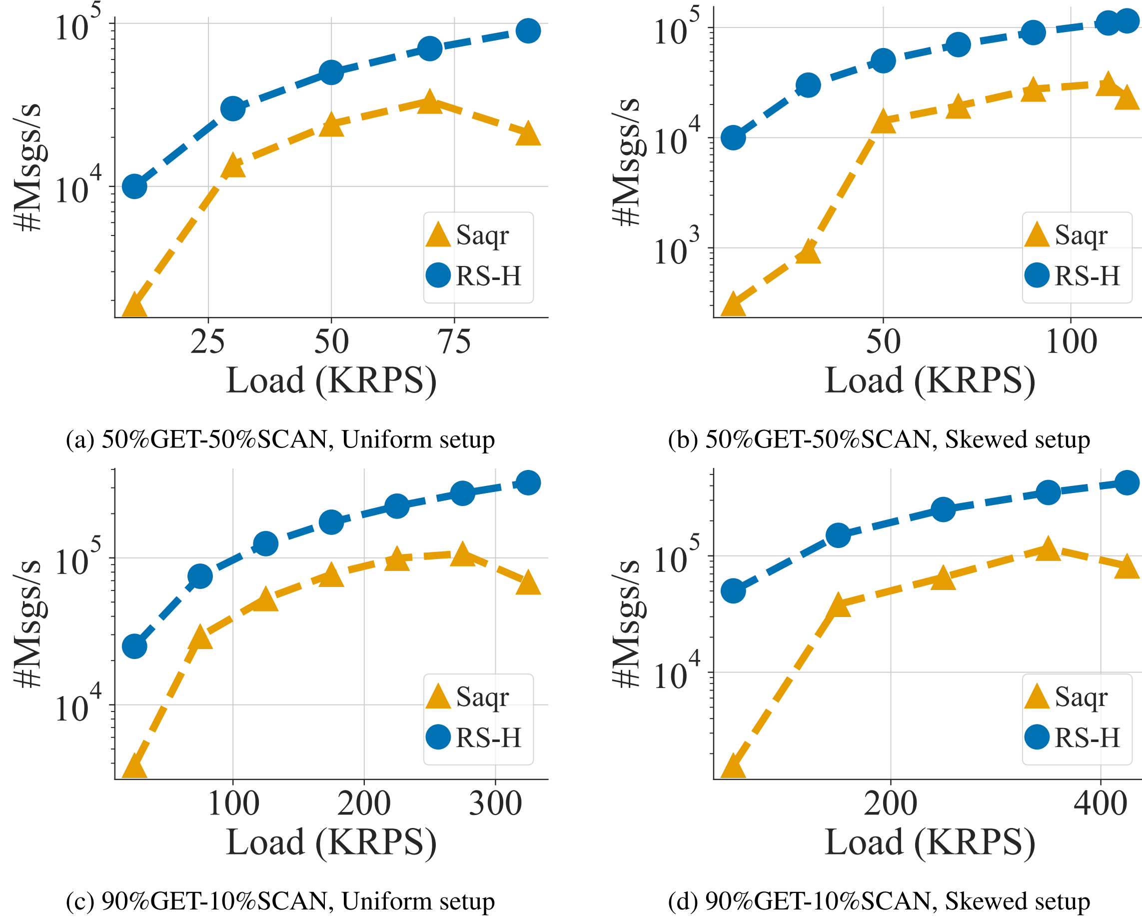


(a) 50%-GET, 50%-SCAN



(b) 90%-GET, 10%-SCAN

Figure 5.4: Response time in the Skewed worker placement.



(a) 50%GET-50%SCAN, Uniform setup

(b) 50%GET-50%SCAN, Skewed setup

(c) 90%GET-10%SCAN, Uniform setup

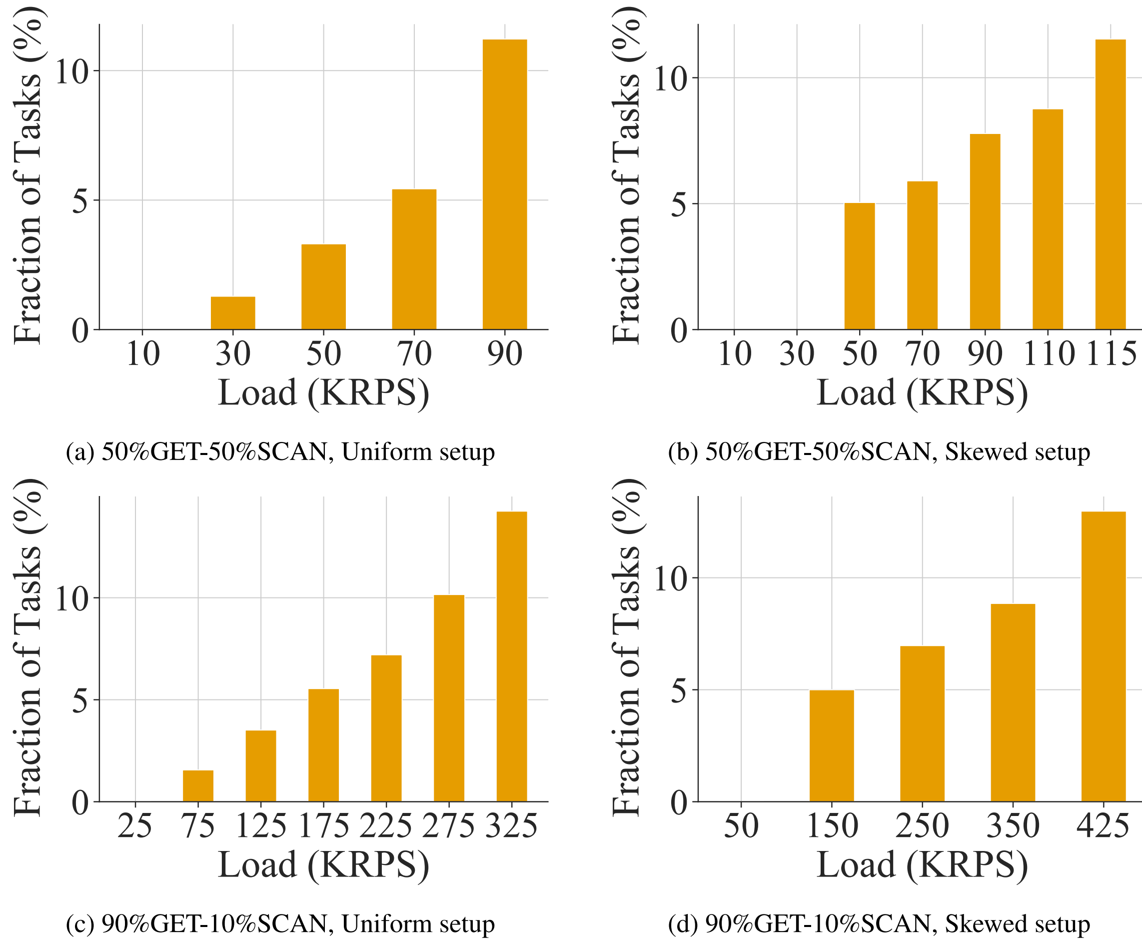
(d) 90%GET-10%SCAN, Skewed setup

Figure 5.5: Rate of state updates.

We also note that RS-R does not exchange update messages, because it randomly selects racks. It, however, significantly increases the response time (Figures 5.3 and 5.4).

Processing Overheads. A Saqr scheduler may selectively resubmit a fraction of task packets to the switch pipeline to update the scheduler view after making a scheduling decision. Figure 5.6 shows the fraction of task packets that are processed twice by a switch in our experiments. The results show that maximum fraction of the task packets that are resubmitted is 14%. When the load is low, the scheduler does not need to resubmit tasks as rate of reply packets is high enough to automatically update the scheduler state. We note that simple solutions to update the state would result in a 100% resubmission rate because they resubmit every packet after scheduling a task.

In addition, we measure the total processing overheads on switches which includes the total number of additional packets (all except the task packets) that are processed by the five switches in our testbed. The processing overheads for Saqr consists of processing the (i) resubmitted packets (including the $idleRemove$ and task packets) and (ii) state update messages from leaf schedulers. Figure 5.7 shows the result for experiments with different worker placement and workloads. As shown in the figures, Saqr consistently reduces the processing overheads on the switches for every



(a) 50%GET-50%SCAN, Uniform setup

(b) 50%GET-50%SCAN, Skewed setup

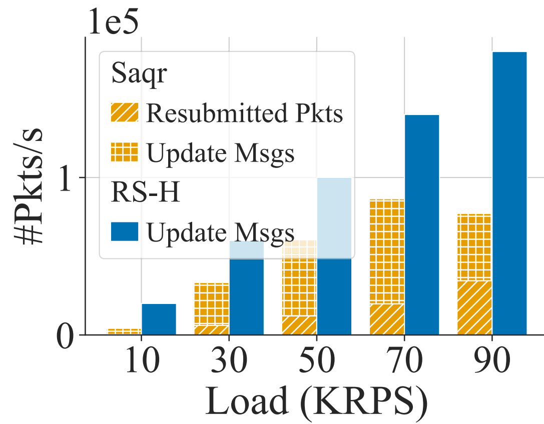
(c) 90%GET-10%SCAN, Uniform setup

(d) 90%GET-10%SCAN, Skewed setup

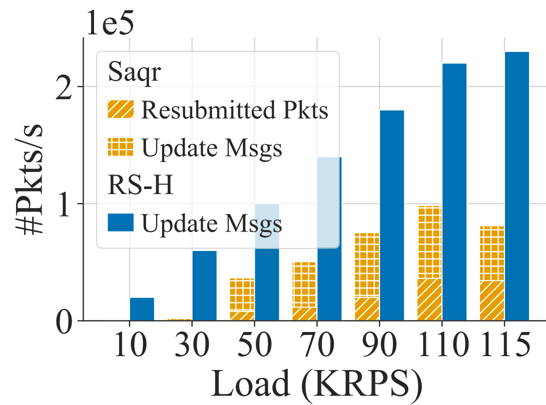
Figure 5.6: Fraction of resubmitted tasks.

load condition. For example, in the Skewed setup with 90% GET and 10% SCAN (Figure 5.7a), Saqr reduces the total processing overheads by up to 31.5X when the system is lightly loaded and up to and 2.5X when the system is heavily loaded.

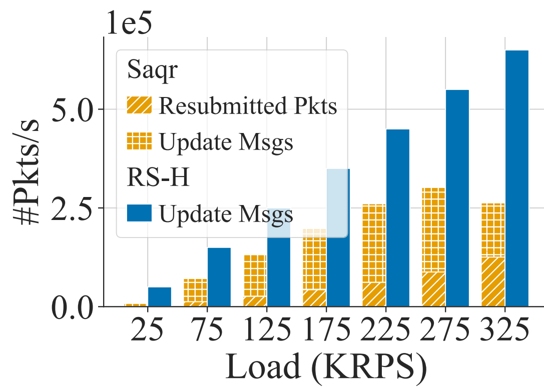
Overall, our results show that Saqr effectively adapts to different conditions in the system to enable low communication and processing overheads without having any prior knowledge about the workloads and arrival rates.



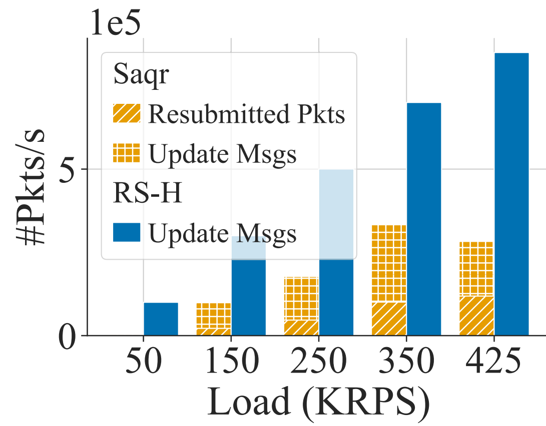
(a) 50%GET-50%SCAN, Uniform setup



(b) 50%GET-50%SCAN, Skewed setup



(c) 90%GET-10%SCAN, Uniform setup



(d) 90%GET-10%SCAN, Skewed setup

Figure 5.7: Total packet processing overheads.

Chapter 6

Evaluation using Simulation

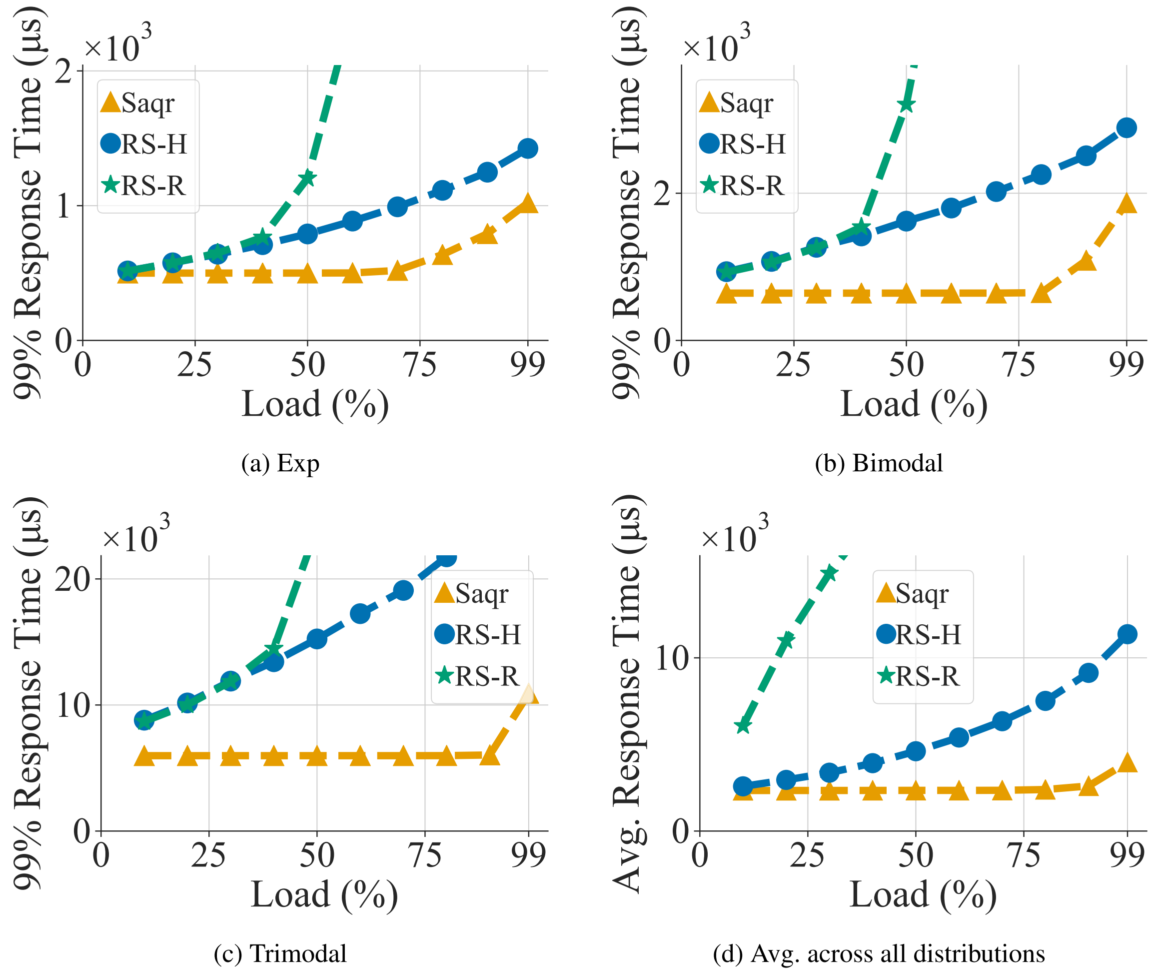
We conduct simulations to evaluate the performance of Saqr versus Racksched [76] in a large-scale datacenter.

6.1 Simulation Setup

Topology and Virtual Clusters. We simulate a network with a multi-rooted Clos topology consisting of 27,648 hosts. Each host has 32 cores and accommodates a maximum of 32 workers (one worker per core). The network has 1,152 spine switches and the same number of leaf switches distributed in 48 fully connected pods. We simulate the operations of 1K concurrent virtual clusters in the datacenter. We use a setup similar to prior works [65, 47] to allocate workers to virtual clusters. The number of workers per virtual cluster follows an exponential distribution with min=50, max=20K and mean=685, where the total number of workers is 685K. Each worker has its private task queue and runs a first-come-first-serve (FCFS) policy to process tasks.

Workloads and Task Arrival Model. To capture the performance in multiple realistic settings, we generate three workloads with different distributions for task processing times: (1) Exp (100) is an exponential distribution with mean=100 μs, which represents a set of tasks with similar processing times such as single type of query for in-memory key-value stores and caching servers [39, 76], (2) Bimodal (50%–50 μs, 50%–500 μs) and (3) Trimodal ($33.\overline{3}$%–50 μs, $33.\overline{3}$%–500 μs, $33.\overline{3}$%–5000 μs), which together simulate request patterns typically observed in a mix of simple and complex processes such as *get/put* and *scan* operations [57]. We run the experiments using each mentioned workload to observe the behaviour of Saqr under different task service time distributions.

We generate requests following a Poisson arrival process. This stresses the system as non-uniform inter-arrival delays generate bursts that can cause temporary queue imbalance and impact the tail latency [60]. We keep increasing the system load (by increasing the arrival rate of requests) until we reach the maximum load for each virtual cluster, which is given by $\lambda = n/\overline{s}$, where n is the number of workers in virtual cluster and $\overline{s}$ is the meas task execution time. In the figures, we report the system load as a percentage of the of maximum load.



(a) Exp
(b) Bimodal
(c) Trimodal
(d) Avg. across all distributions

Figure 6.1: Response time for different workload distributions in our simulations.

6.2 Comparison against the State-of-Art

We compare Saqr versus the two variations of Racksched described in **??** that support multi rack scheduling.

Response Time. In Figure 6.1, we present the tail response time of Saqr and Racksched for each workload distribution. To show the performance overall distributions, we report the average response time across the three workload distributions.

Figures 6.1a to 6.1c depict the tail response time observed by the median virtual cluster. Saqr reduces the tail response time by up to 3X at moderate loads and achieves higher throughput for any given response time value. The gains are more significant when the dispersion in the service times is higher. For the Trimodal distribution, Saqr achieves 5X higher throughput than RS-H when the target tail response time is 10 ms.

Figure 6.1d shows the average task response time across all workload distributions and virtual clusters. The response time of RS-R grows rapidly with load: the average response times become 10

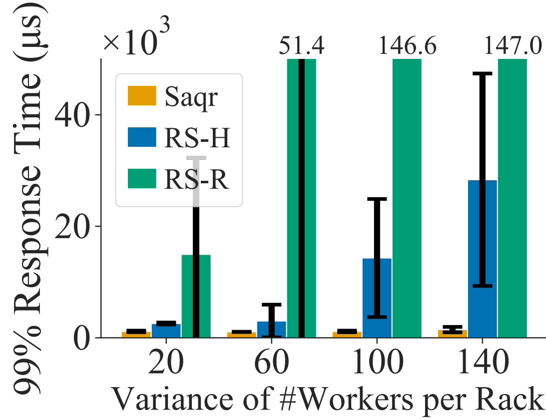


Figure 6.2: Impact of worker placement.

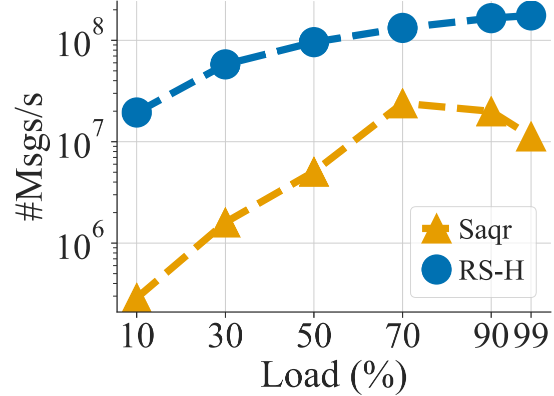


Figure 6.3: Rate of state updates.

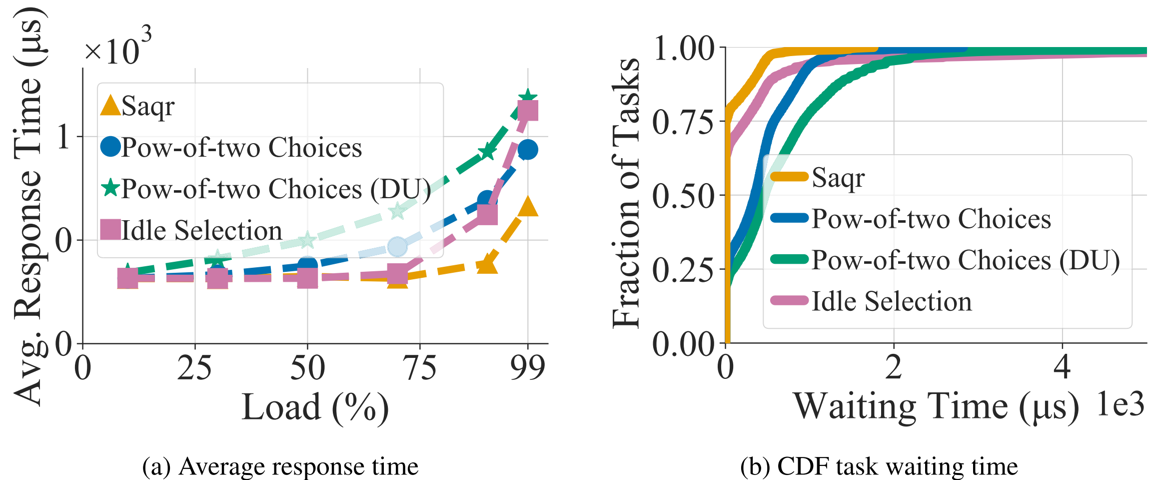
ms and 32 ms at throughput values of 10% and 90%, respectively. Compared to RS-H, Saqr reduces the average response time by up to 71% at throughput of 90%.

To assess the impact of worker distribution imposed by the resource allocation policy and available resources [10], we measure the response time versus the variance in number of workers per rack. A high (low) variance indicates scattered (clustered) workers per virtual cluster. Figure 6.2 shows the average and standard deviation of tail response times (at 90% load) observed by different virtual clusters. Saqr has stable and small response times with low variability even when the workers are scattered across racks. For example, when the workers are scattered (i.e., variance is 140), Saqr reduces the average response times by up to 94% and 99% compared to RS-R and RS-H, respectively.

Communication Overhead. Figure 6.3 shows the rate of state update messages processed by a single spine scheduler for the Exp(100) workload distribution. We observe a similar trend for other distributions. In RS-H, leaf schedulers send the updated average load of a rack after receiving a reply from the workers and the rate linearly increases with the task arrival rate. Leaf schedulers in Saqr, however, selectively send update messages to the spine layer. Thus, Saqr reduces the rates by up to 67X and 15.5X at 10% and 99% loads, respectively.

6.3 Analysis of Saqr

Breaking Down Saqr Benefits. We analyze the contributions of the two components of the proposed scheduling policy: (i) scheduling tasks to idle nodes using idleness information, and (ii) scheduling tasks to busy nodes using power-of-two choices. In this experiment, we focus on five sample virtual clusters with sizes ranging from 50 to 20K workers and use the Bimodal task distribution. We compare Saqr versus schedulers using only idle worker selection or power-of-two choices. In the case of idle node selection, and similar to prior works [49], the scheduler assigns tasks to nodes randomly if it is not aware of any idle nodes. We also simulate two variants of the power-of-two choices scheduling. The schedulers in the first variant immediately update their state



(a) Average response time

(b) CDF task waiting time

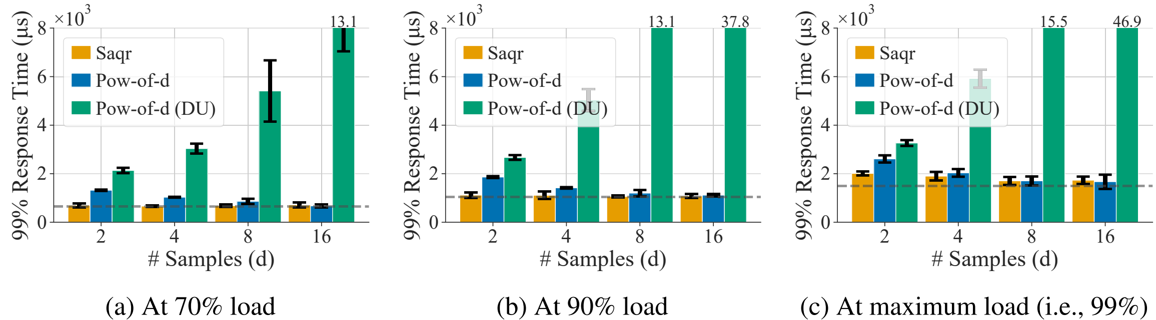
Figure 6.4: Break-down of Saqr performance benefits.

after making a decision. In the second variant, the schedulers rely only on the reply packets from workers to update their local view, which is referred to as delayed updates (DU) [76].

We present in Figure 6.4a the average response time versus the system load. The results show that the scheduling policy of Saqr outperforms all other policies for all load values. To get a better understanding, we plot the CDF of task waiting times at 90% load in Figure 6.4b. The task waiting time is the time from when a task arrives at a worker until it begins execution. Idle worker selection policy increases the fraction of tasks experiencing zero waiting time by 2X compared to power-of-two choices policies. However, power-of-two policies can significantly reduce the tail waiting times when a scheduler is not aware of any idle nodes. The scheduling policy in Saqr uses both idleness and load information to reduce the task waiting time. In addition, as shown in the figures, relying on delayed updates increases both the average response time and tail waiting time at every load condition because the policy has no fresh view of workers.

Impact of Number of Samples on Scheduling Decisions. Saqr schedulers use power-of-d choices to schedule a task on less loaded workers when the scheduler is not aware of any idle worker. We evaluate the response time when increasing the d value. The setup for this experiment is similar to the previous experiment. We simulate schedulers that only use power-of-d choices policy. In addition, we simulate a centralized hypothetical *Oracle* scheduler that tracks the real-time load of *every single worker in the entire datacenter*. The Oracle scheduler selects the worker with the minimum number of pending tasks for each arriving task.

Figure 6.5 shows the results at different load values. Saqr policy is less impacted by the number of samples compared to the power-of-d choices algorithm. For example, in Saqr, at 90% system load (Figure 6.5b), setting d to 16 improves the tail response times up to 13% compared to $d = 2$. Also, when $d = 2$, Saqr is able to achieve the same performance as power-of-d choices with $d = 16$.



(a) At 70% load (b) At 90% load (c) At maximum load (i.e., 99%)

Figure 6.5: Impact of number of samples on scheduling decisions for different policies. Dotted line shows the performance of an *Oracle* scheduler.

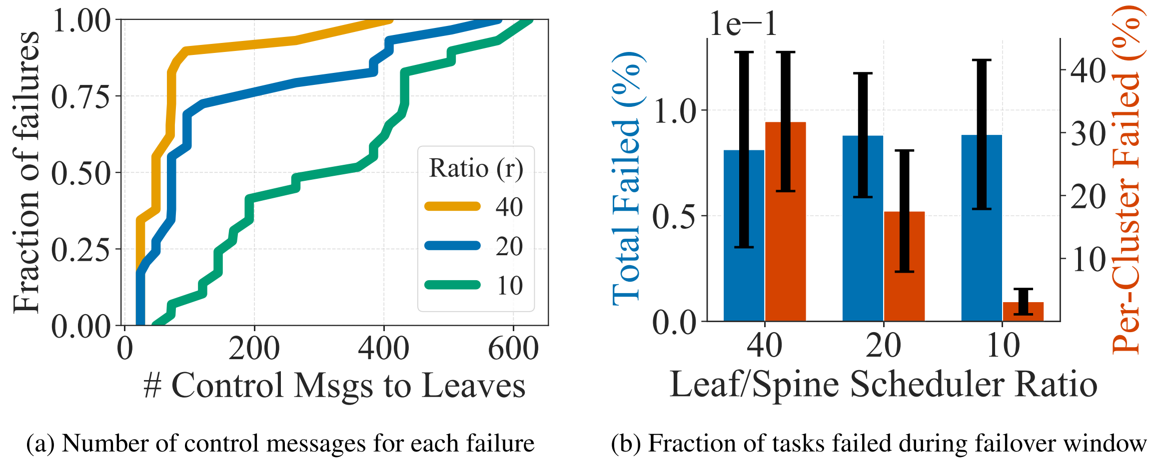
Our results show that when system load is less than or equal to 90%, Saqr achieves comparable performance to the *Oracle* scheduler using only two samples. When the load is maximum (Figure 6.5c), the tail response time for our solution is 34% larger than the Oracle.

The results show that increasing the number of samples with delayed state updates significantly degrades the performance. These results are in line with both testbed and simulation results from previous works [76, 22]. This is because with more samples, schedulers tend to send multiple consecutive tasks to the worker that *apparently* has least load until the load gets updated by the worker. This results in system instability where schedulers keep overloading different workers.

Overall, our simulation results show that Saqr requires small d values because of its policy that uses idle information and load values efficiently. Setting small d values is important for in-network schedulers. Because of the limited programmability and computing resources, increasing the number of samples inside the data plane requires maintaining additional copies of load arrays in the limited memory (one access per register array) and additional computation stages ($log(d)$ sequential comparisons). Furthermore, avoiding delayed updates inside the switches with large d values is not trivial and requires additional bandwidth and resources (as discussed in §3.2.3).

Impact of Scheduler Failures. We analyze the impact of spine scheduler failures where all of the 1K virtual clusters are running. Upon detecting a failure event, the controller sends the event to clients that were using the spine scheduler. We use a trace-driven delay model for the latency of a control message from the centralized controller to the clients (based on [34]). The simulator places clients and the centralized controller in randomly-selected racks and decides the one-way delay from the controller to client depending on the number of hops. The experiment is repeated 30 times, each time we fail one random spine scheduler. Burst spine failures are rare and median time between failures is multiple hours [32]; therefore, we only consider single spine failures.

We use the value *r* as the ratio of leaf to spine schedulers to control the number of spine schedulers per virtual cluster. When r equals the number of racks in a virtual cluster, the load state for the racks is maintained at only one spine scheduler. As we decrease r, the state will be partitioned among more spine schedulers. For example, the value $r = 40$ indicates using fewer spine schedulers compared to when $r = 10$. Each failed switch in our experiments is in charge of scheduling



(a) Number of control messages for each failure

(b) Fraction of tasks failed during failover window

Figure 6.6: Impact of spine switch failures on the applications.

for 2–8 virtual clusters. Figure 6.6 shows the impact of spine switch failures and the trade-off for using different number of spine schedulers for clusters. Figure 6.6a shows the number of messages sent from the centralized controller to the leaf switches as a result of the failure. When the state is distributed among more spine schedulers (i.e., small r values), a failure can result in larger number of control messages. This is because the Saqr controller sends a message to each leaf switch that needs to update its state. In the worst case ($r = 10$), an average of 306 (maximum 624) messages need to be sent for each spine failure event. The centralized controllers in today's datacenters can send thousands of updates per second [53, 74].

Figure 6.6b shows the fraction of failed scheduling requests during the failure, i.e., before all impacted clients use another spine scheduler. The left-axis shows the fraction of total scheduling requests and the right-axis shows fraction of scheduling requests per each virtual cluster that were using the failed spine as scheduler. In the worst-case, when using fewer spine schedulers (i.e., large r values), Saqr can handle more than 99.2% of scheduling requests of the datacenter under a spine switch failure. Using more spine schedulers provides better availability for each cluster (right-axis) during the failure. When $r = 10$, more than 97% of tasks of each cluster are handled by other schedulers during the failure.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we presented the design, implementation, and evaluation of Saqr, a datacenter-wide in-network task scheduler for multi-tenant datacenters. In contrast to traditional schedulers, Saqr offloads the scheduling to network switches, which enables scheduling tasks at high rates. Saqr distributes the load information of workers among leaf and spine schedulers. We proposed a new scheduling policy that tracks the load information of workers and minimizes the task response time. We presented multiple ideas and data structures to efficiently realize the scheduling policy in programmable switches. We also designed methods to propagate updated load values among schedulers. We implemented Saqr in a testbed with a modern programmable switch and compared its performance against Racksched [76], the state-of-art in-network scheduler. We also evaluated the performance of Saqr using large-scale simulations. Our experimental and simulation results showed that Saqr is scalable and robust against failures, and it substantially outperforms Racksched across all performance metrics.

Part of our contributions addressed the limitations of state-of-the-art programmable switch technology for offloading applications to the network. We presented new ideas and data structures to realize Saqr in the data plane. We believe that the design principles and high-level ideas for efficiently tracking worker states and scheduling the tasks using multiple switches inside the network will remain useful in the future despite the potential technological advancements. Our work takes a step forward towards distributed in-network computing and realizing scalable low-latency task scheduling. With technological advancements in the switch hardware capabilities in future, the simplicity of Saqr would allow multiple applications to be offloaded to the network at the same time. Saqr can also benefit from more memory and processing resources on the switches to scale to larger clusters or realize more complex scheduling policies inside the switches.

7.2 Future Work

The work presented in this thesis can be extended in multiple directions. One important aspect that can be extended is realizing other scheduling policies inside the network. For example, fast task migrations between worker queues can be realized using the state that is stored inside the switches. This could improve the performance in heavily dispersed workloads by avoiding the head-of-line blocking in the worker queues. Using similar state distribution mechanisms as presented in this thesis, the mentioned extensions can be realized by identifying the idle workers and moving the tasks from the queues of the other workers to idle workers. From a holistic perspective, the in-network scheduler can be integrated with the application-layer schedulers to support a more diverse set of applications hosted in today's datacenters. The in-network scheduler can be helpful to provide high throughput and low latency for a certain type of tasks while the application-layer scheduler can be used for more complex decision making processes such as handling inter-task and locality constraints (e.g., the tasks for virtual cluster A must not run on racks with tasks for virtual cluster B), and different resource sharing policies. Designing such architecture and carefully offloading parts of the scheduling to the network is an interesting research problem that would benefit a broader range of applications in datacenters.

Bibliography

[1] Placement groups - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html. [Online; accessed October 2021].

[2] Data sharing on traffic pattern inside facebook's data center network. https://bit.ly/3tLgIqz, Jan 2017. [Online; accessed October 2021].

[3] Fault tolerance through optimal workload placement. https://bit.ly/2VMR6wQ, September 2020. [Online; accessed October 2021].

[4] Racksched Git Repository. https://github.com/netx-repo/RackSched, 2020. [Online; accessed October 2021].

[5] Apache Lucene search engine. https://lucene.apache.org/, 2021. [Online; accessed October 2021].

[6] Memcached key-value store. http://memcached.org/, 2021. [Online; accessed October 2021].

[7] Redis in-memory data structure store. https://redis.io/, 2021. [Online; accessed October 2021].

[8] RocksDB. https://rocksdb.org/, 2021. [Online; accessed October 2021].

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM'08*, page 63–74, Seattle, WA, USA, August 2008.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty tenants and the cloud network sharing problem. In *Proc. of USENIX NSDI'13*, pages 171–184, Lombard, IL, April 2013.

[11] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *Proc. of USENIX NSDI'20*, pages 667–683, Santa Clara, CA, February 2020.

[12] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[13] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proc. of USENIX OSDI'12*, pages 335–348, Hollywood, CA, October 2012.

[15] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proc. of ACM SIGCOMM'12*, pages 431–442, Helsinki, Finland, August 2012.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[17] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[18] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *Proc. of USENIX ATC'18*, pages 645–650, Boston, MA, July 2018.

[19] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized load balancing with general service time distributions. In *Proc. of ACM SIGMETRICS'10*, pages 275–286, New York, NY, June 2010.

[20] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proc. of ACM ASPLOS'19*, pages 107–120, Providence, RI, April 2019.

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[22] Michael Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1033–1047, 2000.

[23] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proc. of ACM SOCC'16*, pages 497–509, Santa Clara, CA, October 2016.

[24] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC'15*, pages 499–510, Santa Clara, CA, July 2015.

[25] Christina Delimitrou. *Improving resource efficiency in cloud computing*. Stanford University, 2015.

[26] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of ACM ASPLOS'13*, pages 77–88, Houston, Texas, USA, March 2013.

[27] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ACM ASPLOS'14*, pages 127–144, Salt Lake City, Utah, February 2014.

[28] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110, 2015.

[29] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of USENIX NSDI'17*, pages 363–376, Boston, MA, March 2017.

[30] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proc. of USENIX OSDI'20*, pages 281–297, 2020.

[31] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proc. of EuroSys'18*, pages 1–13, Porto, Portugal, April 2018.

[32] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of ACM SIGCOMM'11*, pages 350–361, Toronto, Canada, August 2011.

[33] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of USENIX OSDI'16*, pages 99–115, Savannah, GA, USA, November 2016.

[34] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM'15*, pages 139–152, London, United Kingdom, August 2015.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI'11*, pages 22–22, Boston, MA, March 2011.

[36] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proc. of ACM SIGCOMM'15*, pages 435–448, London, United Kingdom, August 2015.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP'17*, pages 121–136, Shanghai, China, October 2017.

[38] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[39] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proc. of USENIX NSDI'19*, pages 345–360, Boston, MA, February 2019.

[40] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC'15*, pages 485–497, Santa Clara, CA, July 2015.

[41] D. Katz and D. Ward. Bidirectional forwarding detection (bfd). RFC 5880, June 2010.

[42] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. In *Proc. of EuroP4'20*, pages 7–12, Barcelona, Spain, December 2020.

[43] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *Proc. of USENIX ATC'19*, pages 863–880, Renton, WA, July 2019.

[44] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[45] Collin Lee and John Ousterhout. Granular computing. In *Proc. of ACM HotOS'19*, pages 149–154, Bertinoro, Italy, May 2019.

[46] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. of ACM SOSP'17*, pages 104–120, Shanghai, China, October 2017.

[47] Xiaozhou Li and Michael J Freedman. Scaling ip multicast on datacenter topologies. In *Proc. of ACM CoNEXT'13*, pages 61–72, Santa Barbara, CA, December 2013.

[48] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[49] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011.

[50] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of ACM SIGCOMM '17*, pages 15–28, Los Angeles, CA, August 2017.

[51] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198, 2019.

[52] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[53] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proc. of ACM SIGCOMM'09*, pages 39–50, Barcelona, Spain, August 2009.

[54] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of USENIX ATC'13*, pages 219–230, 2013.

[55] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proc. of ACM SOSP'13*, pages 69–84, Farminton, Pennsylvania, November 2013.

[56] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. of ACM EuroSys'18*, pages 1–14, Porto, Portugal, April 2018.

[57] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *Proc. of the VLDB Endowment*, 10(11):1526–1537, 2017.

[58] Benoît Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas Clausen. Stateless load-aware load balancing in p4. In *Proc. of IEEE ICNP'18*, pages 418–423, Cambridge, UK, September 2018.

[59] Dan RK Ports and Jacob Nelson. When should the network be the computer? In *Proc. of HotOS'19*, pages 209–215, Bertinoro, Italy, May 2019.

[60] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proc. of ACM SOSP'17*, pages 325–341, Shanghai, China, October 2017.

[61] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. When to hedge in interactive services. In *Proc. of USENIX NSDI'21*, pages 373–387, April 2021.

[62] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[63] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. of USENIX NSDI'21*, pages 785–808, April 2021.

[64] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. of ACM EuroSys'13*, pages 351–364, Prague, Czech Republic, April 2013.

[65] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM'19*, pages 458–471. Beijing, China, August 2019.

[66] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849*, 2019.

[67] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of USENIX NSDI'12*, pages 225–238, San Jose, CA, April 2012.

[68] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. of ACM CoNEXT'16*, pages 205–219, Irvine, CA, December 2016.

[69] Mark van der Boor, Sem Borst, and Johan van Leeuwaarden. Load balancing in large-scale systems with multiple dispatchers. In *Proc. of IEEE INFOCOM'17*, pages 1–9, Atlanta, GA, May 2017.

[70] Manohar Vanga, Arpan Gujarati, and Björn B Brandenburg. Tableau: a high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[71] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SOCC'13*, pages 1–16, Santa Clara, California, October 2013.

[72] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proc. of ACM EuroSys'15*, pages 1–17, Bordeaux, France, April 2015.

[73] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proc. of ACM SIGCOMM'18*, pages 176–190, Budapest, Hungary, August 2018.

[74] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proc. of ACM SIGCOMM'17*, pages 295–308, Los Angeles, CA, August 2017.

[75] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. of ACM SIGCOMM'20*, pages 126–138, Virtual Event, NY, USA, August 2020.

[76] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proc. of USENIX OSDI'20)*, pages 1225–1240, November 2020.