©Copyright 2021
Hanfei Yu

FaaSRank: A Reinforcement Learning Scheduler for Serverless Function-as-a-Service Platforms

Hanfei Yu

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2021

Committee:

Wes J. Lloyd

Athirai A. Irissappane

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

FaaSRank: A Reinforcement Learning Scheduler for
Serverless Function-as-a-Service Platforms

Hanfei Yu

Chair of the Supervisory Committee:
Assistant Professor Wes J. Lloyd
School of Engineering and Technology

In recent years, Function-as-a-Service (FaaS) platforms have gained popularity as a way to deploy serverless applications in the cloud instigating the rise of serverless computing. Current serverless FaaS platforms generally use simple, classic scheduling algorithms for distributing function invocations while ignoring FaaS characteristics such as rapid changes in resource utilization and the freeze-thaw life cycle. In this thesis, we present FaaSRank, a scheduling service for serverless FaaS platforms based on information monitored from servers and functions. FaaSRank automatically learns scheduling policies through experience using reinforcement learning (RL) and neural networks. To bridge the gap between FaaS scheduling and RL techniques, we develop a novel Score-Rank-Select architecture for FaaSRank.

We implement FaaSRank in Apache OpenWhisk, an open source serverless FaaS platform. We evaluate FaaSRank against other baseline schedulers including OpenWhisk's default scheduler on two 13-node OpenWhisk clusters by adapting real-world serverless workload traces provided by Microsoft Azure. Our results show that FaaSRank minimizes the overall average function completion time by 9.25% and 10.10% over the default scheduler. This improvement is realized while reducing the average number of inflight function invocations by 59.62% and 70.43% freeing computational resources compared to the default scheduler for our experiments on two clusters, respectively.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables iv

Chapter 1: Introduction 1
- 1.1 Contributions 3
- 1.2 Thesis Organization 3

Chapter 2: Background and Motivation 4
- 2.1 FaaS and Function Scheduling 4
- 2.2 Static-rank Scheduler 5
- 2.3 An Illustrating Example 6
- 2.4 Deep Reinforcement Learning 8

Chapter 3: Overview 10
- 3.1 Challenges 10
- 3.2 Objective 11

Chapter 4: Design 13
- 4.1 Server assessment 14
- 4.2 Score function 14
- 4.3 Training FaaSRank 18

Chapter 5: Implementation 20
- 5.1 OpenWhisk architecture 20
- 5.2 FaaSRank 22

Chapter 6: Evaluation 23
- 6.1 Baseline schedulers 23
- 6.2 Experimental setup 24

6.3 Results 26
Chapter 7: Related Work 33
Chapter 8: Conclusion 35
Glossary 36
Bibliography 38

LIST OF FIGURES

Figure Number | Page

2.1 Illustration of a typical FaaS platform. 4
2.2 Overall average FCT of three schedulers 7
4.1 The embedding of a state observed by FaaSRank. 13
4.2 The policy network in FaaSRank. 14
5.1 The workflow of FaaSRank scheduling for OpenWhisk. 21
6.1 Workload invocation traces for OpenWhisk clusters. 26
6.2 Overall average FCT of schedulers measured on two OpenWhisk clusters . . . 27
6.3 The time series of inflight invocations recorded on two OpenWhisk clusters . 30
6.4 Percentage of cold starts for individual schedulers on two OpenWhisk clusters 31

LIST OF TABLES

Table Number Page

2.1 Formulas used in Static-rank scheduler 6

4.1 Resource utilization metrics observed by FaaSRank 15

4.2 Server and function metrics observed by FaaSRank 16

6.1 Characterizations of Experimental Serverless Function Workloads 23

6.2 Normalized percentage (%) of average FCT of individual functions relative to Hashing scheduler on Canada Cloud cluster (**C** Canada Cloud cluster, **H** Hashing, **SR** Static-rank, **RR** Round-robin, **LC** Least-connections, **G** Greedy) 28

6.3 Normalized percentage (%) of average FCT of individual functions relative to Hashing scheduler on AWS cluster (**A** AWS cluster, **H** Hashing, **SR** Static-rank, **RR** Round-robin, **LC** Least-connections, **G** Greedy) 29

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to his advisors, Dr. Wes J. Lloyd and Dr. Athirai A. Irissappane. He also would like to thank Amazon Web Services and Dr. Hao Wang for kindly offering experimental resources. Last but not least, the author would like to thank his parents and colleagues for their help and support.

Chapter 1

INTRODUCTION

Serverless computing is a new cloud computing paradigm that has growing prosperously in recent years. Function-as-a-Service (FaaS), the most commonly used service delivery model of serverless computing, has become increasingly popular [10]. Serverless FaaS platforms free users from low-level tasks while automating resource provisioning, scaling, and isolation. Users are only responsible for deploying source code and configuring memory limits for applications. FaaS platforms feature pay-as-you-go pricing models while enabling simplified deployment of applications. Consequently, FaaS platforms provide an enticing option for developers to consider for hosting computational workloads that simplify management with potential to reduce costs incurred from renting idle servers. Major cloud providers offer FaaS platforms such as AWS Lambda [7], Microsoft Azure Functions [35], Google Cloud Function [14], and IBM Cloud Functions [20].

Similar to traditional web service load balancing, FaaS platforms schedule function invocations by distributing requests to multiple worker servers for execution. Classic load balancing algorithms have been working well for traditional request dispatching on clusters hosting web services. Some commonly used load balancing algorithms include: (1) Round-robin, one of the simplest and most used algorithms which distributes client requests to servers in rotation. Round-robin assumes all the servers are identical when scheduling. (2) Least-connections, a dynamic algorithm where client requests are distributed to the server with the least number of active connections at the time the client request is received. Least-connections takes active load into consideration when scheduling. (3) Greedy, an algorithm that always sends client requests to the same server until reaching its capacity. Greedy improves resource utilization of servers while introducing contention. (4) Hashing, provides an algorithm that generates a unique hash key for each client to subsequently always distribute requests from the client to the same server. Hashing guarantees that requests from

the same client are assigned a similar execution environment. Providers generally employ one or a combination of classic algorithms to balance load for web services.

Though sharing some characteristics with traditional web service scheduling, FaaS scheduling introduces new challenges. First, traditional schedulers are not FaaS aware. On a traditional web service cluster, service deployments are typically fixed and do not change. On a FaaS cluster, function invocations are bursty and stateless while also experiencing the infrastructure freeze-thaw life cycle [29]. Several minutes after finishing execution, temporary infrastructure of inactive FaaS functions known as function instances [49] are removed from a server. When function instances are recreated, they may not be provisioned on the same server. This inconvenience can result in additional initialization overhead as necessary source code and libraries may not be cached on the new host, a phenomenon recognized as "latency of cold starts" [22]. FaaS function deployment dynamically changes the free capacity on the cluster, whereas traditional web service clusters feature largely static deployment and resource management. Second, a FaaS cluster enables all resources for unused functions to be reclaimed: e.g processes, memory, disk space, and CPU capacity. As resources can be repurposed, FaaS clusters are more dynamic and changing in nature, *i.e.*, conditions on individual servers are more likely to change rapidly. It's non-trivial to provide an intelligent scheduling approach that simultaneously addresses all of the challenges unique to FaaS scheduling. New scheduling approaches are needed to address these challenges to expand upon traditional load balancing algorithms for function scheduling in FaaS platforms.

However, existing FaaS platforms have generally adopted simple, classic scheduling algorithms while ignoring unique challenges associated with FaaS characteristics (*e.g.*, AWS Lambda [7] and Apache OpenWhisk [3]). In this thesis, we present FaaSRank, a scheduling service designed for serverless FaaS platforms. FaaSRank learns dynamic scheduling policies through experiences using reinforcement learning (RL) and neural networks. Given a high-level goal (*e.g.*, average function completion time in our context), FaaSRank leverages active resource information monitored from servers and functions to automatically make decisions on scheduling function invocations. FaaSRank optimizes the overall performance of FaaS workloads while maintaining scheduling fairness of improving individual functions from different clients. To bridge the gap between FaaS scheduling problems and off-the-shelf

RL techniques, such as training reusable neural networks for scalable clusters, we develop a novel Score-Rank-Select architecture to ensure FaaSRank works well for modern FaaS platforms.

1.1 Contributions

Here are the primary contributions of the thesis:

- We propose the design of FaaSRank, a general RL-based function scheduling algorithm for serverless platforms.
- We implement a prototype of FaaSRank in Apache OpenWhisk [3].
- We then implement four baseline schedulers for comparison purposes including the novel FaaS-aware Static-rank scheduler, as well as three classic schedulers: Round-robin, Greedy, and Least-connections.
- We evaluate FaaSRank using ten realistic serverless functions collected from [34] [53] and [47]. We adapted real-world serverless traces from Microsoft Azure Functions [46] to create experimental workloads to enable our extensive evaluations of FaaSRank performance against five baseline schedulers.

1.2 Thesis Organization

The remaining chapters of the thesis are organized as follows: Chapter 2 reviews the background and motivates the need for FaaSRank. Chapter 3 introduces the challenges and objectives. Chapter 4 presents the design and training of FaaSRank. Chapter 5 presents the details of our FaaSRank implementation. Chapter 6 describes the extensive experiments we conduct to evaluate FaaSRank and other baselines. Chapter 7 reviews related research, and Chapter 8 provides our conclusions.

Chapter 2
BACKGROUND AND MOTIVATION

In this chapter, we introduce general FaaS platforms and existing function scheduling strategies. We use an example evaluated on a realistic FaaS platform to motivate the need for an intelligent scheduler. We also briefly illustrate how to learn scheduling policies using reinforcement learning.

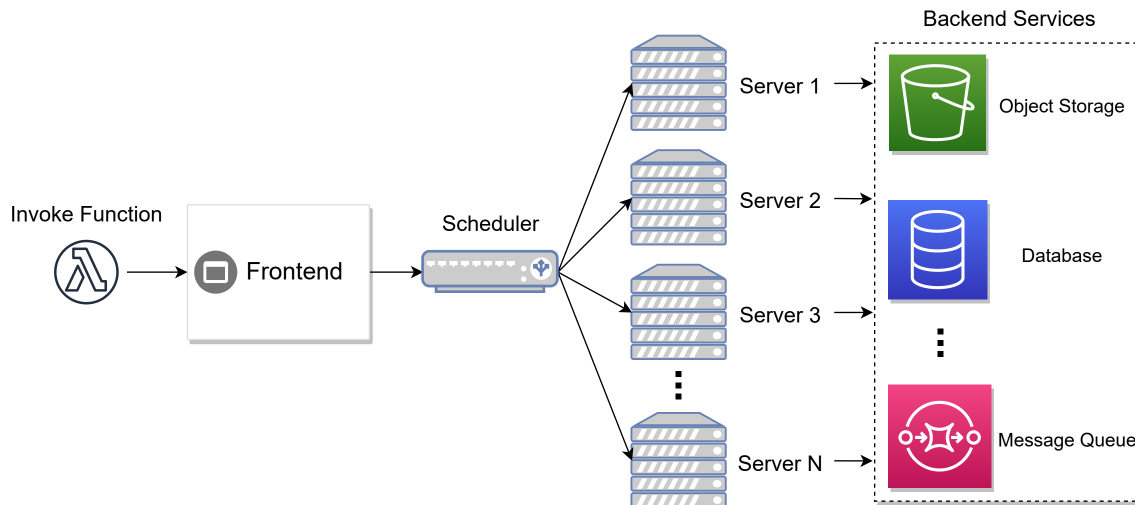


Figure 2.1: Illustration of a typical FaaS platform.

2.1 FaaS and Function Scheduling

Figure 2.1 describes a general architecture of FaaS platform. The frontend receives function invocations and forward them to a scheduler. The scheduler maintains a list of available servers for function execution. Once being informed of an incoming invocation, the scheduler selects a server to schedule and execute the function. Servers in the cluster may have access to various backend services such as object storage, databases, and message queues. When

a user invokes a FaaS function and a server is chosen, the server fetches the input of the function from object storage, executes the function, and stores the output in object storage after execution. A database then keeps records of function invocations as a log.

Function scheduling algorithms vary for existing FaaS platforms. Open-source FaaS platforms generally use classic algorithms for function scheduling. For example, Apache OpenWhisk [3] adopts a hashing method to schedule functions within a distributed cluster. Kubernetes leverages classic scheduling algorithms to place containers including round-robin, least-connection, and hashing [25]. Details regarding scheduling algorithms implemented by commercial FaaS platforms are not available publicly, nevertheless researchers reveal some facts through reverse engineering. [49] and [26] to identify that AWS Lambda [7] greedily packs containers running function invocations on Virtual Machines (VMs) to improve resource utilization.

Existing approaches for function scheduling, *e.g.*, classic algorithms or the packing strategy employed by AWS Lambda, tend to introduce performance problems. For classic algorithms, though hashing increases the possibility of avoiding cold starts, *i.e.*, the delay incurred by launching a new function instance with its dependencies [22], can suffer from resource contention when co-located with too many other functions on the same server. In this case, round-robin and least-connections are unaware of function dependencies, which can result in more cold starts. AWS Lambda's packing strategy also introduces resource contention, and hence results in performance degradation [49]. Given available metrics of servers, it's we select a server to schedule a function invocation? Chapter 2.3 illustrates that human-designed strategies are conservative and can hardly handle complicated FaaS scenarios.

2.2 Static-rank Scheduler

We first investigated the development of heuristic-based scheduling algorithms for FaaS platforms. We developed the Static-rank scheduler, which calculates an overall score for each server using a fitness function with fixed weights that manually set the importance of each factor's contribution to the overall score. Our Static-rank scheduler considers resource utilization, infrastructure states, and load conditions for assessing a server. The overall

score is calculated as:

$$\begin{aligned}\textbf{score} = 2 \times cpu + 1.5 \times mem + disk + net + cpu_load_avg \\ + infrastructure_state + available_memory\end{aligned} \tag{2.1}$$

where 2 and 1.5 are weights assigned to CPU and memory attributes. The Static-rank scheduler computes the fitness function for each server and then selects the server with the highest overall score to schedule the function invocation.

Specifically for resource utilization attributes *cpu*, *mem*, *disk*, and *net*, we use the formulas from Table 2.1 to aggregate the individual metrics described in Table 4.1. The *cpu_load_avg* is the Linux system load average representing the average system load over a period of time. The *infrastructure_state* describes whether the function's runtime environment is cold (0) or warm (1). The *available_memory* represents an individual server's free memory for deploying functions reported by the FaaS framework measured in MB. We normalize each of the metrics before calculating the individual scores. We describe our *resource utilization metrics* used to assess servers in more detail in Chapter 4.1.

Attribute	**Formula**
`cpu`	Δcpu_user + Δcpu_nice + Δcpu_kernel + Δcpu_idle + Δcpu_iowait + Δcpu_irq + Δcpu_softirq + Δcpu_steal + Δcpu_ctx_switches
`mem`	1.5 × memory_free + memory_buffers + memory_cached
`disk`	Δdisk_read_count + Δdisk_read_merged_count + Δdisk_read_time + Δdisk_write_count + Δdisk_write_merged_count + Δdisk_write_time
`net`	Δnet_byte_recv + Δnet_byte_sent

Table 2.1: Formulas used in Static-rank scheduler

2.3 An Illustrating Example

To motivate the need of an intelligent scheduler for FaaS platforms, we conduct an evaluation of function scheduling on Apache OpenWhisk [3], an open-source FaaS platform used

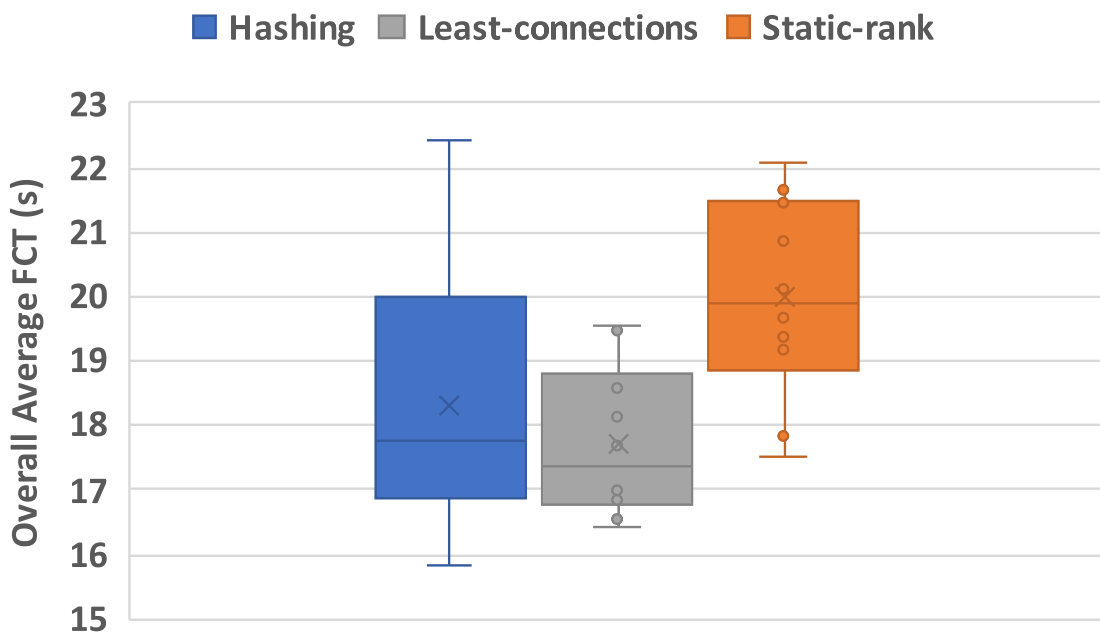


Figure 2.2: Overall average FCT of three schedulers

to implement the commercial IBM Cloud Functions FaaS platform [20], to demonstrate performance of adopting different schedulers. We deployed an OpenWhisk cluster with 10 servers available for executing function invocations known as invokers. Each invoker has 8 vCPU cores and 16 GBs of memory, with 2 GBs available for function runtimes. The details of our OpenWhisk configuration are described in Chapter 6.2.

For this motivating experiment, three schedulers are examined:

- **Hashing scheduler.** The default scheduler employed by OpenWhisk. The hashing scheduler calculates a hash value for each function, and always schedules the same function invocation to the same server with the aim of maximizing warm starts.

- **Greedy scheduler.** We use a simple greedy algorithm to mimic the scheduling strategy adopted by AWS Lambda, which always schedules functions to the same server until the server no longer has available resources. The greedy scheduler maximizes the

resource utilization of in-use servers while trying to activate as few servers as possible.

- **Static-rank scheduler.** A heuristic scheduler described in Chapter 2.2.

We next evaluate the three schedulers identified above by adapting workloads sampled from real-world serverless function traces provided by Microsoft Azure [46]. We generated a 60-second workload based on the Azure traces that captures a series of FaaS function requests over a 60-second period for ten distinct functions. The details of our experimental workload traces are described further in Chapter 6.2.

Figure 2.2 reports the overall average function completion time (FCT) achieved by our three schedulers observed over 10 iterations. The Least-connections scheduler outperforms our other two schedulers, reducing the overall average FCT by 3.28% and 11.27%, compared to the Hashing scheduler and our Static-rank scheduler respectively. As the weights of Static-rank's fitness function are fixed, it cannot adapt to unique workload characteristics. Our example illustrates that our Static-rank scheduler in fact performs ***worse*** than two other classic schedulers. However, given the nature of serverless workloads, manually devising a reasonable fitness function for scheduling may require exhaustive trial-and-error to determine good fitness function weights appropriate for various for workloads.

In contrast to Static-rank, we present FaaSRank, a self-learning scheduling algorithm to schedule functions for FaaS platforms based on deep reinforcement learning (DRL). FaaSRank uses neural networks to automatically learn good function scheduling policies based on high-level objectives such as average FCT. FaaSRank evaluates resource utilization metrics of individual servers and information of the incoming function invocation, ranks all the servers, and selects the best server to schedule the function.

2.4 Deep Reinforcement Learning

FaaSRank uses RL and neural networks to learn function scheduling algorithms for FaaS. In a general RL setting, an agent learns how to benefit most from making sequential decisions by iteratively interacting with the environment and accumulating knowledge from previous experience. RL is well-suited to learning policies for computer systems, because RL

agents are able to learn from real-world workloads and operating conditions without human-designed inaccurate assumptions. RL has been applied to various scheduling problems, such as resource management [32], network optimization [11], and device placement [36].

Specifically in RL, at every time t, the agent first observes a state s_t of the environment, and then makes a decision on taking an action a_t. Following the action, the environment changes its state to s_{t+1} and the agent perceives a reward r_t as feedback. The interactions are stochastic and assumed to be a Markov process, *i.e.*, the next state s_{t+1} and reward r_t solely depend on the previous state-action pair s_t, a_t. Thus the agent learns to maximize its expected cumulative rewards

$$\mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t * r_t\Big], \tag{2.2}$$

where $\gamma \in (0, 1]$ is the discount factor to discount the sum of rewards by how far off in the future they're obtained [39].

The agent takes actions based on *policy*, defined as a mapping between states and actions. A policy π outputs an action a_t when given a state s_t, *i.e.*, $a_t \sim \pi(\cdot|s_t)$. It's common to use *function approximators* to represent parameterized policies. A function approximator outputs computable functions that depend on a set of adjustable parameters, θ, which we can adjust to affect the behavior of a policy via optimization algorithms. We refer to θ as *policy parameters* and represent the policy as $a_t \sim \pi_\theta(\cdot|s_t)$. In DRL, neural networks are used as function approximators to solve stochastic RL tasks, as neural networks are end-to-end differentiable for training and self-adaptive without hand-crafted features [33]. Therefore we use neural networks to represent the scheduling policy of FaaSRank.

Chapter 3
OVERVIEW

FaaSRank is an intelligent scheduler that uses neural networks to make function scheduling decisions for serverless platforms. On FaaS platforms, scheduling events involves orchestrating where a function executes on a distributed cluster. Activated by an event, FaaSRank takes as an input the current state information of the cluster, and the function request, and outputs a scheduling action, *i.e.*, a server to schedule the incoming function invocation.

3.1 Challenges

We tackle three key challenges by designing FaaSRank:

1. **Server Assessment**. Given a function invocation request, the scheduler must select the best server from a FaaS cluster for function scheduling. It's non-trivial to compose together available metrics to assess individual servers to make reasonable trade-offs between cold starts and resource contention in real-time.

2. **Cluster Scalability**. It's empirical to define a fixed output size of a neural network, *i.e.*, a fixed number of servers within a cluster. However, it's common for multiple servers to join or leave the cluster a problem that can force neural networks to be retrained.

3. **Huge action space.** Providers host commercial FaaS platforms on clusters consisting of thousands of servers. Selecting a server from a huge cluster requires the scheduler agent to be trained over a huge action space, *i.e.*, the output size of a neural network must be linear to the cluster size. Mapping conditions to thousands of actions poses a challenge for training the scheduler, which has to explore the action space to learn a good policy.

To address Challenge 1, we adopt a combination of *resource utilization metrics* and function information as features to characterize the state of individual servers for function execution. To address Challenge 2 and 3, we propose a *score function* inspired by [33], which is implemented using neural networks to make scheduling decisions across clusters having an arbitrary number of servers. We describe our proposed solutions in detail in Chapter 4.

3.2 Objective

FaaSRank optimizes the overall average *Function Completion Time (FCT)* of a workload. The FCT of a function invocation is defined as the time from its arrival until completion. This includes initialization overhead, waiting time in any platform queues, and execution time.

We consider a FaaS platform that handles a multiple function concurrent workload. Let S denote the set of functions invoked within the workload, f denotes a function invocation in S. The platform captures the FCT c_f of an invocation after it completes execution. The total FCT C of a workload is denoted by $C = \sum_{f \in S} c_f$, which we want to minimize. Hence, we aim to minimize the overall average FCT given by:

$$\bar{C} = \frac{\sum_{f \in S} c_f}{\sum_{f \in S} 1}. \tag{3.1}$$

Note that this objective does not guarantee scheduling fairness of individual functions, *i.e.*, FaaSRank treats a workload as an entity to optimize the overall average FCT, but does not guarantee a performance improvement for every function. Other RL-based scheduling approaches share similar objectives with FaaSRank. DeepRM achieves scheduling fairness by assuming the ideal completion time and ideal resource consumption of jobs, *i.e.*, DeepRM assumes that jobs execute identically regardless of the state of resource contention in system [32]. Decima also optimizes the average job completion time for a workload without addressing fairness [33].

We assessed the scheduling fairness provided by FaaSRank and report our results in Chapter 6. We compare the average FCT of individual functions processed by FaaSRank and compare with other baseline scheduling approaches. Our results show that in additional

to the overall average FCT of the workload, FaaSRank is able to achieve the best average FCT for most of the individual functions while maintaining good performance for others.

Chapter 4

DESIGN

In this chapter, we first present the design of FaaSRank. Specifically, we present our approaches to address the challenges identified in Chapter 3: server assessment (Chapter 4.1), cluster scalability and huge action space (Chapter 4.2). We also describes the algorithm used to train FaaSRank (Chapter 4.3).

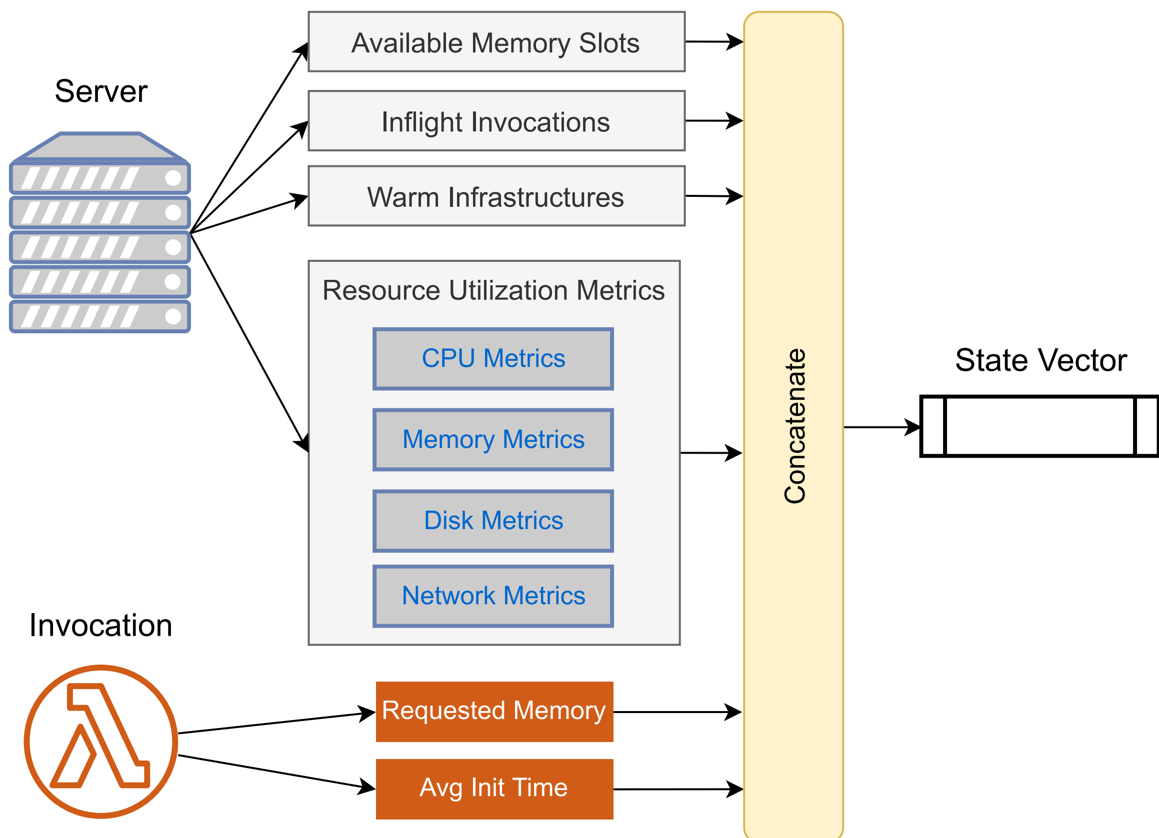


Figure 4.1: The embedding of a state observed by FaaSRank.

4.1 Server assessment

FaaSRank uses *resource utilization metrics* to assess the resource condition of individual servers. Table 4.1 shows the resource utilization metrics that FaaSRank collects from each server, where Δ indicates the change in resource utilization for the sampling interval. We characterize four dominant types of resource utilization when assessing a server: CPU, memory, disk and network I/O. Previous research has shown that resource utilization metrics are powerful in identifying resource contention of Infrastructure-as-a-Service (IaaS) cloud [28] [27] [18], and predicting performance and costs of cloud workloads [13]. Other than resource utilization, FaaSRank also adopts additional useful metrics to assess conditions regarding cluster load and infrastructure as shown in Table 4.2.

For each state observation, FaaSRank collects the resource utilization metrics from each server, and encapsulates them with other metrics shown in Table 4.2. Figure 4.1 describes the embedding of a state observed by FaaSRank, which contains information of a server and the incoming function invocation. FaaSRank concatenates all the information into a flat feature vector as input to the score function in Chapter 4.2.

4.2 Score function

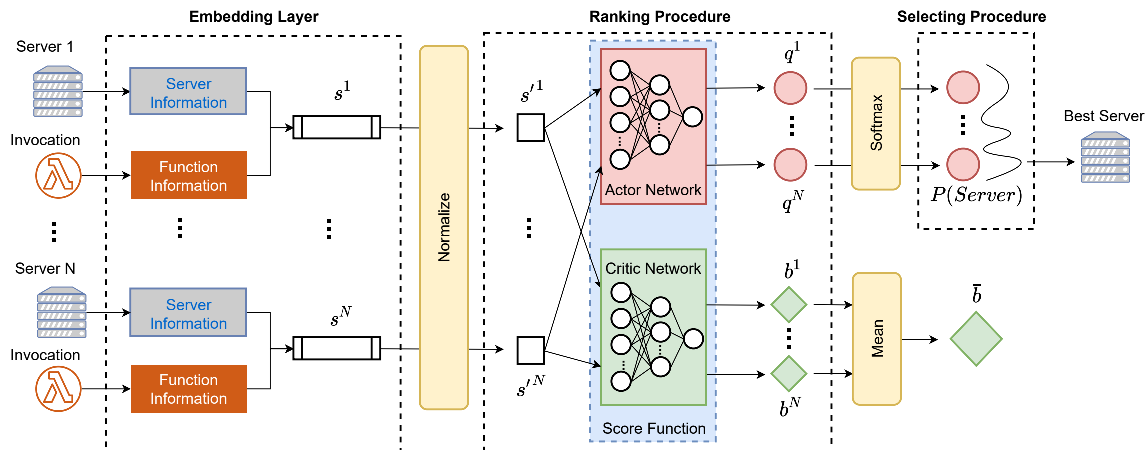


Figure 4.2: The policy network in FaaSRank.

Metric	Description
Δcpu_user	CPU time in user mode
Δcpu_nice	CPU time executing prioritized processes
Δcpu_kernel	CPU time in kernel mode
Δcpu_idle	CPU idle time
Δcpu_iowait	CPU time waiting for I/O to complete
Δcpu_irq	CPU time servicing HW interrupts
Δcpu_softirq	CPU time servicing soft interrupts
Δcpu_steal	CPU time spent by other operating systems
Δcpu_ctx_switches	Number of context switches
cpu_load_avg	Average system load over the last minute
memory_free	Physical RAM left unused by the system
memory_buffers	Temporary storage for raw disk blocks
memory_cached	Physical RAM used as cache memory
Δdisk_read	Number of disk reads completed
Δdisk_read_merged	Number of disk reads merged together
Δdisk_read_time	Time spent reading from the disk
Δdisk_write	Number of disk reads completed
Δdisk_write_merged	Number of disk writes merged together
Δdisk_write_time	Time spent writing
Δnet_byte_recv	Network Bytes received
Δnet_byte_sent	Network Bytes written

Table 4.1: Resource utilization metrics observed by FaaSRank

Metric	Description
`available_memory`	Memory available in the server
`inflight_invocations`	Number of inflight requests in the server
`warm_infrastructures`	Warm infrastructures in the server
`requested_memory`	Memory requested by the function
`init_time`	Measured function cold initialization time

Table 4.2: Server and function metrics observed by FaaSRank

FaaSRank calculates a *score function* to rank each server. The server with the highest score is selected for function scheduling. Intuitively, FaaSRank learns how to score individual servers using the common score function to select a server with the highest score, rather than training a neural network to simply choose a specific server, which requires a fixed size cluster resulting in a huge action space. Figure 4.2 visualizes the policy network of FaaSRank and shows how FaaSRank selects the best server given a batch of state information. At time t to schedule an invocation event, the FaaS cluster has in total N available servers. FaaSRank collects a batch of the latest state vectors $s_t = (s_t^1, \ldots, s_t^n, \ldots, s_t^N)$ from the cluster, where n represents the nth available server. After collecting state vectors, FaaSRank normalizes the state batch to ${s'}_t = ({s'}_t^1, \ldots, {s'}_t^n, \ldots, {s'}_t^N)$ as inputs to the score function, which is the core functionality of FaaSRank. The score function is implemented using two neural networks, an actor and a critic network. Actor-Critic methods are effective in reducing training variance and delivering faster convergence [24]. Specifically,

- **Actor network** computes a score q_t^n, which is a scalar value mapped from the normalized state vector ${s'}_t^n$ representing a priority to select the server n. Then FaaSRank applies a Softmax operation [9] to the scores $(q_t^1, \ldots, q_t^n, \ldots, q_t^N)$ to compute the probability of selecting server n based on the priority scores, given by

$$P_t(server = n) = \frac{\exp(q_t^n)}{\sum_{n=1}^{N} \exp(q_t^n)}, \tag{4.1}$$

  at time t.

- **Critic network** outputs a baseline value b_t^n for server n, the averaged baseline value $\bar{b}_t$ is calculated as

$$\bar{b}_t = \frac{\sum_{n=1}^{N} b_t^n}{N}, \tag{4.2}$$

which is used to reduce variance when training FaaSRank.

The whole operation of our policy network is end-to-end differentiable.

FaaSRank requires no manual feature engineering for its score function, *i.e.*, nothing is hard-coded in the score function. Through training, FaaSRank automatically learns what is important for computing a priority score given a state vector. More importantly, the design of FaaSRank is lightweight as it reuses the same score function for all servers and all function invocations. We further describe the details of training FaaSRank in Chapter 4.3.

Algorithm 1 FaaSRank Training Algorithm

1: Initial policy (actor network) parameters θ_0 and value function (critic network) parameters ϕ_0
2: **for** episode k = 0, 1, 2, ... **do**
3: Run policy $\pi_k = \pi(\theta_k)$ in the environment until terminating at time T
4: Collect set of trajectories $\mathbb{D}_k = \{\tau_i\}$, where $\tau_i = (s_i, a_i), i \in [0, T]$
5: Compute reward $\hat{r}_t$ via Equation 2.2
6: Compute baseline value $\bar{b}_t$ from critic network via Equation 4.2
7: Compute advantage $\hat{\mathbb{A}}_t = \hat{r}_t - \bar{b}_t$
8: Update actor network by maximizing objective using stochastic gradient ascent:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathbb{D}_k| T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} \mathbb{L}(s_t, a_t, \theta_k, \theta) \tag{4.3}$$

9: Update critic network by regression on mean-squared error using stochastic gradient descent:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathbb{D}_k| T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} (\bar{b}_t - \hat{r}_t)^2 \tag{4.4}$$

10: **end for**

4.3 Training FaaSRank

All implementations of TEX provide the option of **pdf** output, which is all the Graduate School now requires. Even if you intend to print a copy or two of your thesis—the best way to admire it—create a `pdf` anyway. It will print anywhere.

FaaSRank training proceeds in *episodes*. In each episode, a series of client function invocations arrive at the FaaS platform where each requires an action (selection of a server) to be performed by FaaSRank. When all of the function invocations finish, we consider the episode complete. Let T denote the total number of actions in an episode, and t_i denote the wall clock time of the ith action. Similar to [33], we continuously feed FaaSRank with a reward r after FaaSRank takes an action based on the objective (overall average FCT) mentioned in Chapter 3.2. Concretely, we penalize FaaSRank with $r_i = -(t_i - t_{i-1}) * F_i$ after the ith action, where F_i is the number of inflight function invocations in the FaaS system during the interval $[t_{i-1}, t_i)$. By setting the discount factor γ to be 1 in Equation 2.2, the goal of the algorithm is to maximize the expected cumulative rewards given by

$$\mathbb{E}\Big[\sum_{i=1}^{T} -(t_i - t_{i-1}) * F_i\Big], \tag{4.5}$$

which is aligned with Equation 2.2. Notice that this cumulative objective is exactly the total FCT of a workload, and hence FaaSRank learns to minimize the overall average FCT in Equation 3.1 for a given workload.

FaaSRank uses a policy gradient algorithm for training. Policy gradient methods [48] are a class of RL algorithms that learn policies by performing gradient ascent directly on the parameters of neural networks, denoted by θ, using the rewards received during training. When updating policies, a large number of steps may deteriorate the performance, while a small number of steps may worsen the sampling efficiency. We use the Proximal Policy Optimization (PPO) algorithms, proposed by Open-AI [45], to ensure FaaSRank takes appropriate steps when updating its policies. Recall in Chapter 2.4, π_θ denotes a policy with parameters θ, a is the action taken when observing state s, PPO algorithm updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathbb{E}}_{s,a \sim \pi_{\theta_k}} \Big[\mathbb{L}(s, a, \theta_k, \theta)\Big], \tag{4.6}$$

where $\mathbb{L}$ is the *surrogate advantage* [44], a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy. Specifically we use the PPO-clip version of a PPO algorithm, where $\mathbb{L}$ is given by

$$\mathbb{L}(s, a, \theta_k, \theta) = \min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ g(\epsilon, A^{\pi_{\theta_k}}(s, a))\Big), \tag{4.7}$$

and $g(\epsilon, A)$ is a clip operation defined as

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0, \end{cases} \tag{4.8}$$

where A is the advantage calculated as rewards r subtracted by baseline values b.

In Equation 4.8, ϵ is a hyperparameter which restricts how far the new policy is allowed to go from the old. Intuitively, the algorithm sets a cap for the range of policy updates, so that it's worthless for the new policy to go too far (either positive or negative) from the old, thus ensuring an appropriate range of update steps.

Algorithm 1 presents the training of FaaSRank. During training for each server, the actor network outputs a score, and the critic network outputs a baseline value. For each episode, we record the whole set of trajectories including the states, actions, rewards, baseline values predicted by critic network, and the logarithm probability of the actions for all events. After each training episode finishes, we use the collected information to update the actor and critic networks.

Chapter 5

IMPLEMENTATION

FaaSRank provides a broadly applicable function scheduling algorithm for use in serverless platforms. For concreteness, we describe its implementation in the context of Apache OpenWhisk. Apache OpenWhisk is an open source, distributed serverless platform that executes functions in response to events at any scale, which manages the infrastructure, servers, scaling, and execution of functions using Docker containers [3]. In this chapter, we briefly describe the architecture of OpenWhisk, and then illustrate the workflow of FaaSRank, *i.e.*, how FaaSRank interacts with and makes scheduling decisions for OpenWhisk.

5.1 OpenWhisk architecture

Figure 5.1 describes the architecture of our FaaSRank integrated with OpenWhisk. OpenWhisk exposes an NGINX-based [37] REST interface for creating new functions, invoking functions, and querying results of invocations. Invocations are triggered by users via an interface, and then forwarded to the Controller. The Controller selects an Invoker (typically hosted using VMs) to schedule the function invocation. The Load Balancer inside the Controller makes scheduling decisions for function based on (1) a hashing algorithm, and (2) information from the Invokers, such as health, available capacity, and infrastructure state. Once an Invoker is chosen, the Controller sends the function invocation request to the selected Invoker via a Kafka-based [2] distributed message broker. The Invoker receives the request and executes the function using a Docker container. After the function execution is finished, the Invoker submits the results to a CouchDB-based [1] Database and informs the Controller. Then the Controller returns the results of function executions to users either synchronously or asynchronously.

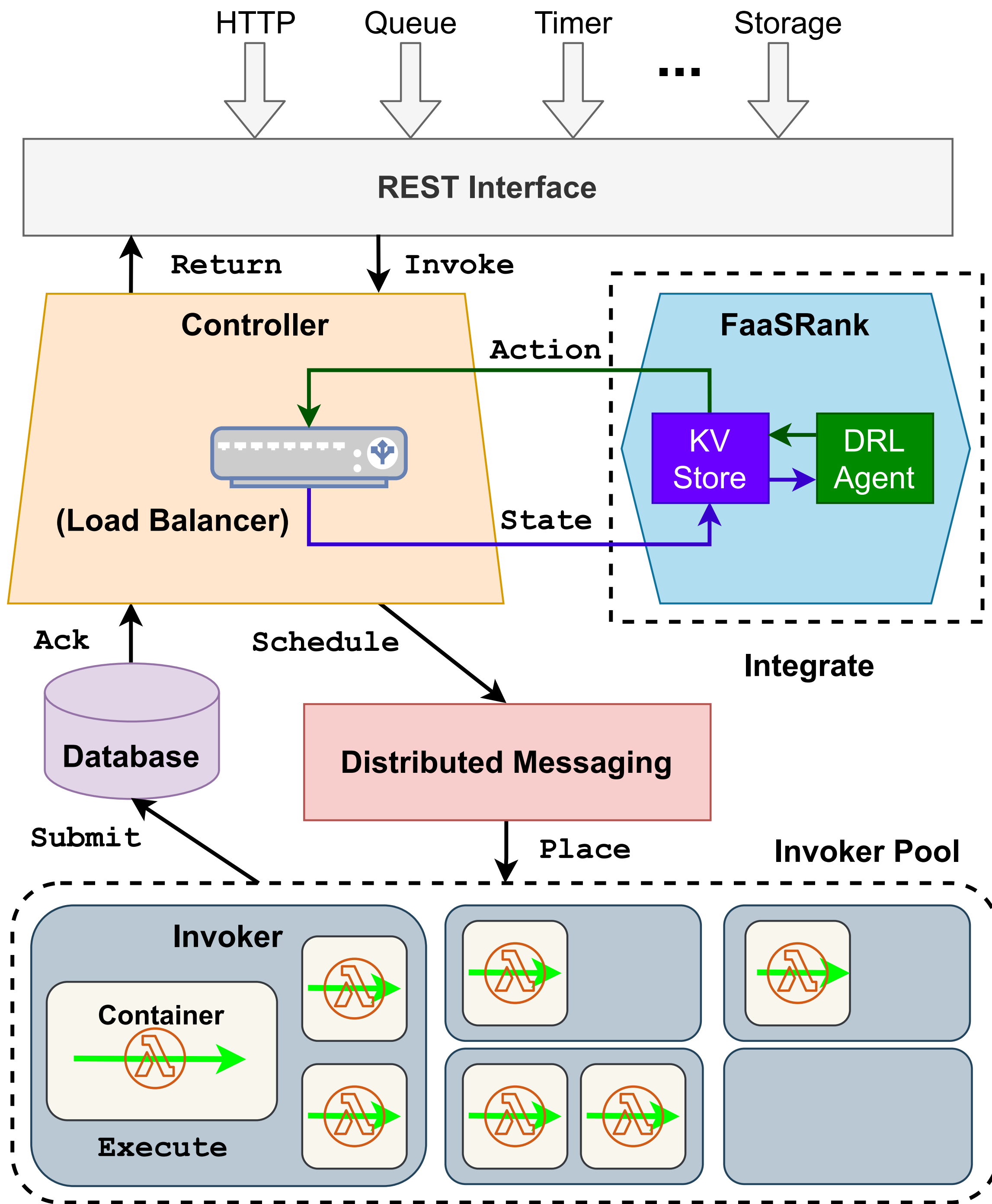


Figure 5.1: The workflow of FaaSRank scheduling for OpenWhisk.

5.2 FaaSRank

Workflow. FaaSRank communicates with the Controller of OpenWhisk via a Key-Value (KV) Store, which is implemented using Redis [42]. Specifically, when receiving a function invocation, the Load Balancer in the Controller first sends the current state information to the KV Store. The DRL Agent in FaaSRank then fetches the state and sends an action to the KV Store, where the Controller picks up the action. Finally the Load Balancer schedules the function invocation on the selected Invoker based on the action provided by FaaSRank.

Implementation. We implement the policy network of our FaaSRank prototype using two neural networks, each with two fully-connected hidden layers. The first hidden layer has 32 neurons and the second layer has 16 neurons, each neuron uses Tanh as its activation function. The agent of FaaSRank is implemented in Python using PyTorch [41]. The implementation of FaaSRank is lightweight as the policy network consists of 2818 parameters (16 KBs in total) because FaaSRank reuses the score function. Mapping a state to a scheduling action takes less than 70 ms.

Training. We use the algorithm presented in Chapter 4.3 to train FaaSRank with 5 epochs per surrogate optimization and a 0.2 clip threshold. We update the policy network parameters using the AdamW optimizer [21] and train FaaSRank for 1000 iterations with a learning rate of 0.0001.

Chapter 6
EVALUATION

Function	Type	Mem (MBs)	Cold (s)	Warm (s)	Init (s)
Dynamic Html (DH)	Web App	512	4.45	2.34	1.55
Email Generation (EG)	Web App	256	2.20	0.21	1.55
Image Processing (IP)	Multimedia	256	5.88	3.52	1.69
Video Processing (VP)	Multimedia	512	6.86	1.19	4.77
Image Recognition (IR)	ML	512	4.28	0.09	1.33
K Nearest Neighbors (KNN)	ML	512	4.99	1.11	3.45
Gradient Descent (GD)	ML	512	4.15	0.60	2.59
Arithmetic Logic Unit (ALU)	Scientific	256	5.72	3.45	1.50
Merge Sorting (MS)	Scientific	256	3.87	1.94	1.54
DNA Visualisation (DV)	Scientific	512	8.57	3.11	4.13

Table 6.1: Characterizations of Experimental Serverless Function Workloads

We conducted extensive evaluations of FaaSRank in the OpenWhisk platform, using realistic serverless functions and real-world serverless traces realised by Microsoft Azure [46].

6.1 Baseline schedulers

For our evaluation, we compare FaaSRank with five other schedulers serving as baselines:

1. **OpenWhisk default hashing scheduler**. The OpenWhisk default scheduler uses a hashing algorithm to schedule function invocations. It calculates a hash value for each function, and always schedules invocations of the same function to the same invoker with the aim of maximizing warm starts.

2. **Round-robin scheduler**. A scheduler that distributes the load by sending successive requests to different invokers in a cyclical manner.

3. **Least-connections scheduler**. A scheduler that always sends the incoming invocation to the invoker with least in-flight requests.

4. **Greedy scheduler**. A scheduler that greedily packs function invocations onto the same invoker until the invoker reaches its capacity. This scheduler mimics the scheduling strategy of AWS Lambda, with the aim of improving resource utilization.

5. **Static-rank scheduler**. A fine-tuned scheduler that also schedules function invocations based on state information that FaaSRank receives, including resource utilization metrics, invoker, and function information. However, in contrast to FaaSRank's trained RL network, the static-rank scheduler employs a fixed, human-designed fitness function (Equation 2.1) to select the best invoker for function invocation.

6.2 Experimental setup

OpenWhisk clusters. We deployed and tested FaaSRank using two independent OpenWhisk deployments on different public clouds, where each OpenWhisk cluster consisted of 13 VMs. One VM hosted the REST front-end, API gateway, and Redis services; one backend VM hosted the Controller, Distributed Messaging, and Database services; one VM hosted our FaaSRank agent; and the remaining 10 VMs were configured as invokers for scheduling functions. We present the details of two OpenWhisk clusters:

1. **Compute Canada Cloud cluster.** We deployed FaaSRank under OpenWhisk on the Compute Canada Cloud [12] using 13 VMs, each with 8 vCPU cores and 32 GBs memory. Each invoker provided 2 GBs RAM for individual function executions while allocating CPU power to functions proportionally based on function memory requirements.

2. **AWS EC2 cluster.** We deployed FaaSRank under OpenWhisk on an AWS EC2 cluster consisting of 13 c5d.2xlarge VMs launched as spot instances, each with 8

vCPU cores, and 16 GBs memory. Each invoker provided 2 GBs RAM for invididual function executions while allocating CPU power to functions proportionally based on function memory requirements.

We trained FaaSRank on our Compute Canada Cloud and AWS OpenWhisk clusters independently. To evaluate scheduler performance, we then repeated our performance experiments 10 times using FaaSRank comparing performance against our five baseline schedulers. We conducted a complete experimental evaluation for each cluster independently.

Functions. For our experiments, we employed ten real-world serverless functions from SeBS [34], ServerlessBench [53] and ENSURE workloads [47]. Table 6.1 characterizes the memory consumption, cold vs. warm runtimes, and initialization overhead for cold starts. All ten functions are implemented in Python3 using CouchDB to store input and output objects. To accurately characterize the functions, we deployed a mini OpenWhisk cluster hosted on an AWS EC2 dedicated host [4], an isolated private server to characterize the average runtime and overhead in a setting without resource contention from other users. The test cluster consisted of 4 c5.2xlarge VMs (user, frontend, backend, and one invoker), each with 8 vCPU cores and 16 GBs memory. We executed each function 10 times to characterize average values for our performance metrics as shown in Table 6.1.

Invocation traces. We randomly selected 20 function invocation traces from the public Azure Functions traces provided in [46]. Our objective was to derive realistic testing workloads that would leverage the full capacity of our two OpenWhisk clusters. We adapted the publicly available traces from Azure by reinterpreting the platform sampling interval from minutes to seconds. This was necessary because the available Azure traces only captured the total number of function invocations per minute. This re-scaling has the effect of increasing the intensity of workload invocations while speeding up our training by reducing the total workload duration. Figure 6.1 depicts the invocation timeline of our two workloads detailing our client function invocations over approximately 60 seconds. To fully occupy cluster capacity we had to increase the intensity of the experimental workload for the AWS EC2 cluster compared to the Compute Canada Cloud cluster. On AWS our trace required 408 function invocations over one minute whereas the Canada Cloud trace required just 292

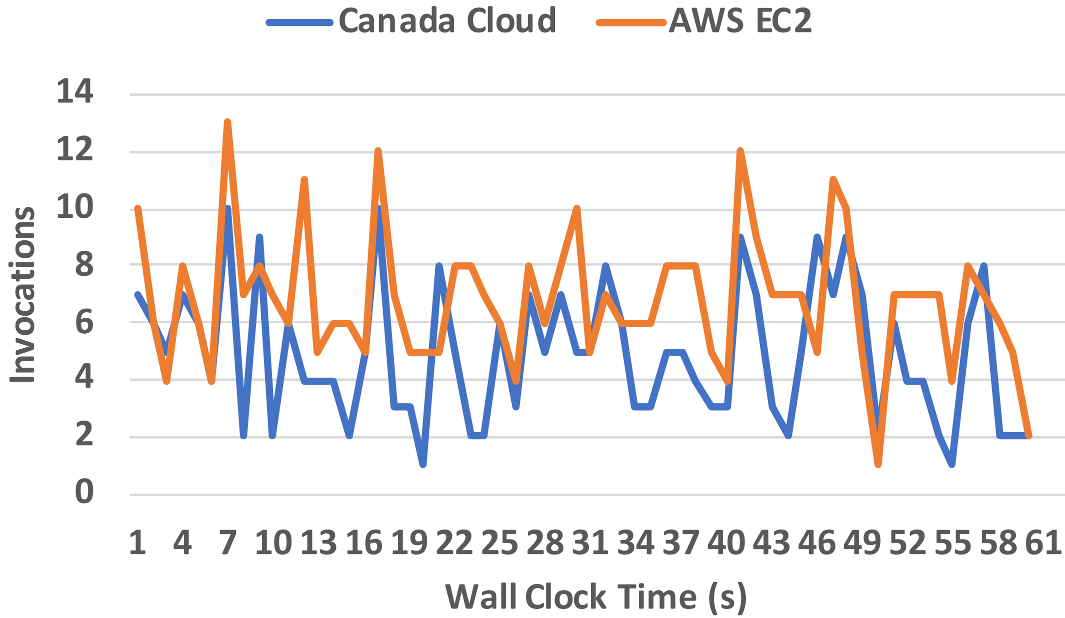
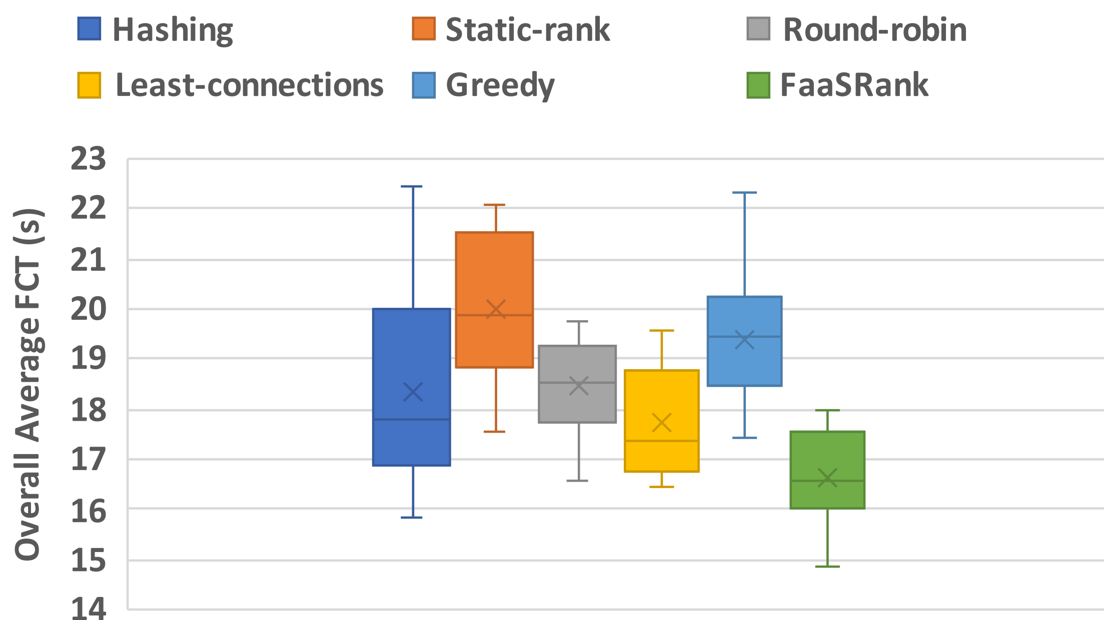


Figure 6.1: Workload invocation traces for OpenWhisk clusters.

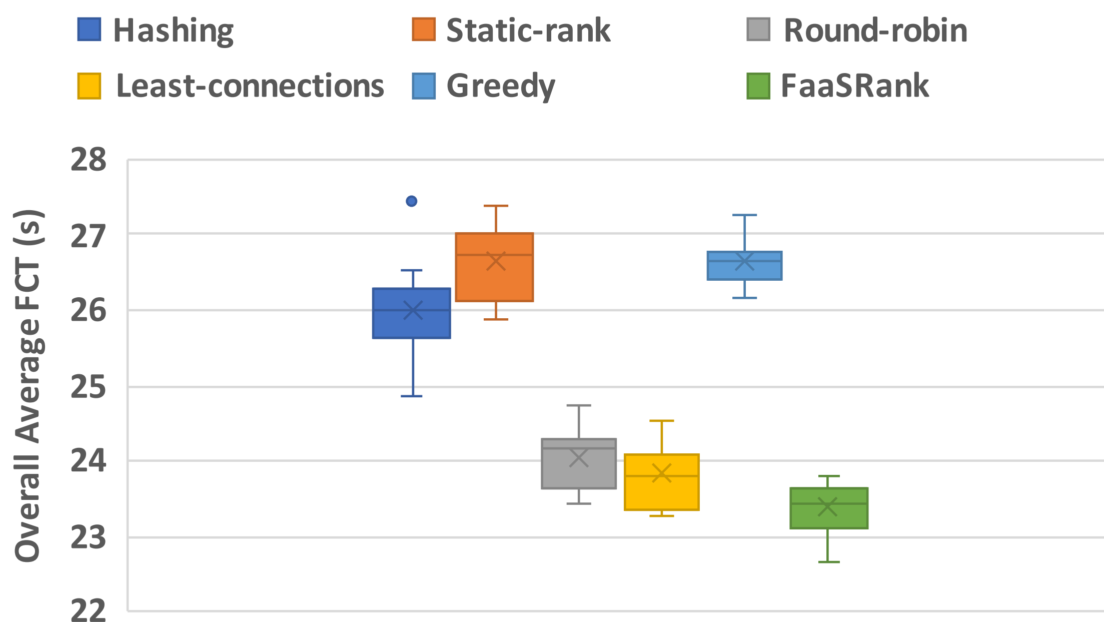
invocations. The AWS EC2 c5 instances were found to be more powerful than the Canada Cloud VMs so we compensated by increasing the workload intensity by approximately 40% for the AWS EC2 cluster.

6.3 Results

We evaluate the FaaS schedulers mentioned in Chapter 6.1 with the following metrics: (1) overall average FCT. This is the primary objective that FaaSRank tries to optimize; (2) fairness. In addition to the overall average FCT, we also want to examine how FaaSRank improves individual average FCT; (3) load. The time series of load condition for an OpenWhisk cluster, which implies the ability of schedulers making scheduling decisions; (4) and percentage of cold starts. A unique but vital metric for evaluating schedulers for FaaS platforms that represents the fraction of function invocations requiring infrastructure initialization relative to the total number of invocations.



(a) Canada Cloud cluster



(b) AWS cluster

Figure 6.2: Overall average FCT of schedulers measured on two OpenWhisk clusters

Overall average FCT. Figure 6.2 shows the overall average FCT of individual schedulers evaluated on two clusters when scheduling the experimental workload. FaaSRank outperforms other five baselines. Compared to OpenWhisk default Hashing scheduler, FaaSRank reduces overall average FCT of the workload by 9.25% and 10.10%, on the Canada Cloud and AWS clusters respectively. Compared to the baseline schedulers, FaaSRank learns to how to reduce overall average FCT while mitigating performance variance.

Function	**H-C**	**SR-C**	**RR-C**	**LC-C**	**G-C**	**FaaSRank-C**
DH	**100.00**	120.97	102.79	108.19	116.18	102.55
EG	**100.00**	125.85	125.33	114.37	116.56	106.89
IP	100.00	117.42	123.07	112.79	127.88	**94.12**
VP	100.00	104.33	97.56	98.23	97.73	**92.27**
IR	100.00	123.16	113.68	**97.21**	117.53	103.85
KNN	100.00	112.53	101.15	91.66	112.31	**82.71**
ALU	100.00	87.81	88.36	79.70	92.33	**75.46**
MS	100.00	99.32	90.50	85.29	94.71	**82.76**
GD	100.00	118.13	102.24	104.42	103.78	**97.69**
DV	**100.00**	122.09	106.81	109.56	115.60	102.86

Table 6.2: Normalized percentage (%) of average FCT of individual functions relative to Hashing scheduler on Canada Cloud cluster (**C** Canada Cloud cluster, **H** Hashing, **SR** Static-rank, **RR** Round-robin, **LC** Least-connections, **G** Greedy)

Fairness. To evaluate the fairness of FaaSRank's scheduling, we recorded the average FCT for each function during the experiment. Table 6.2 and 6.3 present the average FCT of individual functions from 10 repeated runs on the Canada Cloud cluster and AWS EC2 cluster. FaaSRank achieves the minimum average FCT for 6 and 8 functions on the Canada Cloud and AWS clusters respectively. While reducing the average FCT for most functions, FaaSRank also maintained acceptable performance of other functions, *i.e.*, FaaSRank did not hurt the performance of other functions to accelerate others'. FaaSRank achieved the

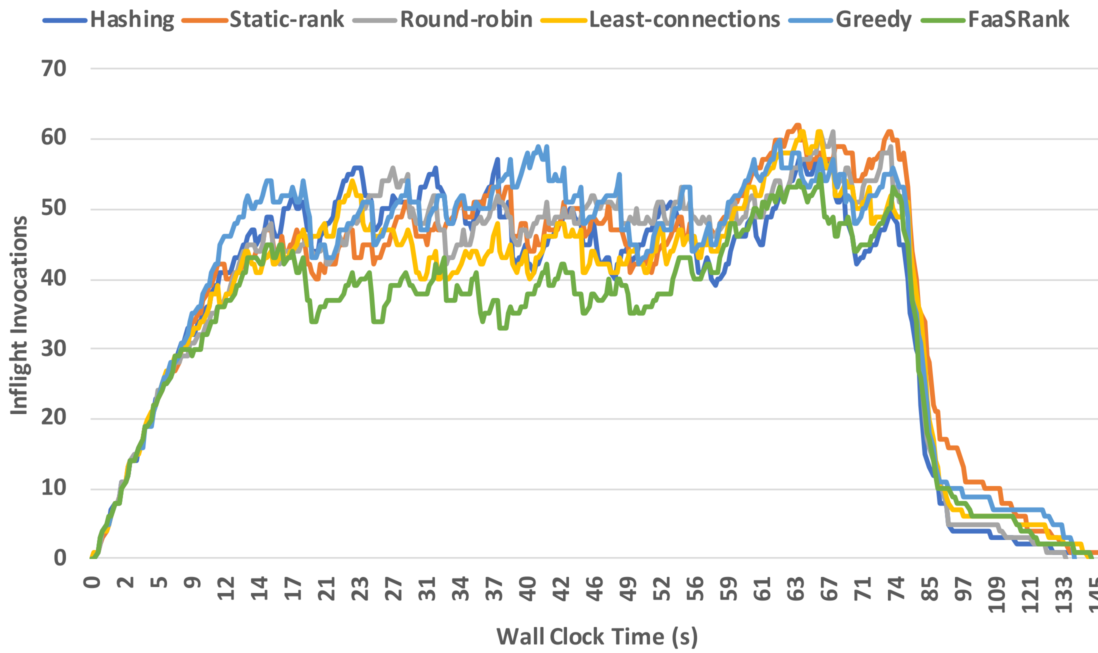
minimum overall average FCT of workloads on the tested schedulers.

Function	**H-A**	**SR-A**	**RR-A**	**LC-A**	**G-A**	**FaaSRank-A**
DH	100.00	104.37	97.83	96.39	105.67	**93.39**
EG	100.00	112.59	96.32	92.20	107.10	**87.99**
IP	100.00	100.17	87.80	86.62	98.10	**83.17**
VP	100.00	101.40	98.68	**92.48**	104.51	93.91
IR	100.00	109.83	97.24	91.23	110.82	**87.75**
KNN	100.00	100.41	90.63	91.84	100.19	**88.60**
ALU	100.00	96.37	88.91	88.21	98.08	**88.19**
MS	100.00	103.60	95.61	93.76	105.44	**90.46**
GD	100.00	106.31	89.44	90.90	102.12	**88.57**
DV	100.00	104.44	93.85	**93.67**	103.91	93.69

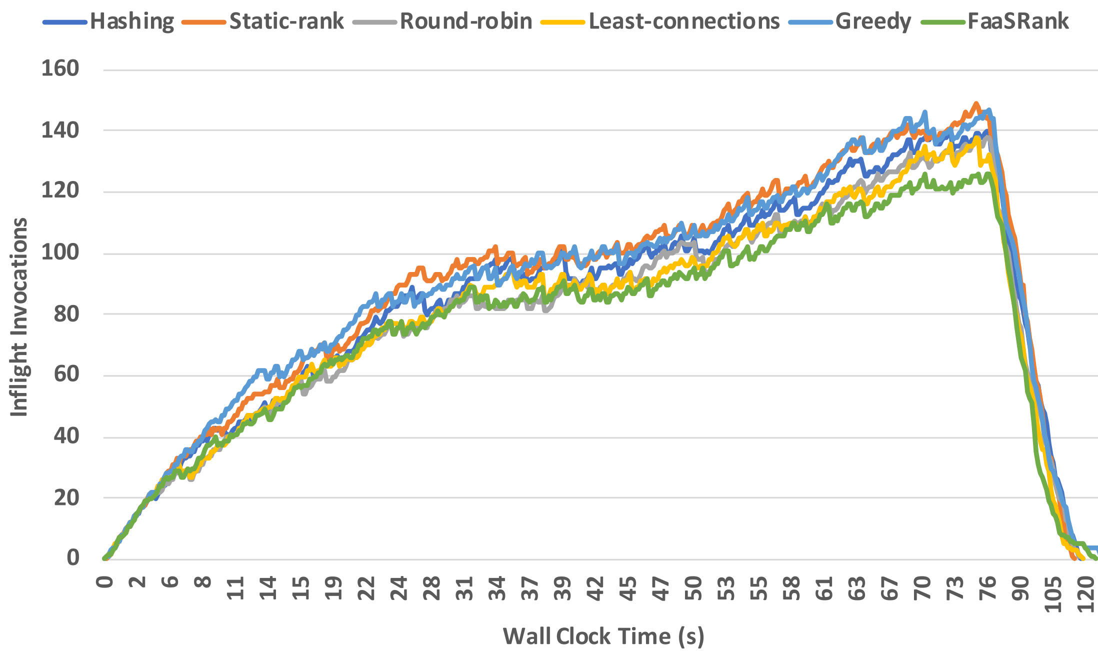
Table 6.3: Normalized percentage (%) of average FCT of individual functions relative to Hashing scheduler on AWS cluster (**A** AWS cluster, **H** Hashing, **SR** Static-rank, **RR** Round-robin, **LC** Least-connections, **G** Greedy)

Load. We recorded the load conditions of each scheduler during the experiments. Figure 6.3 displays the time series of inflight invocations for each scheduler on our two OpenWhisk clusters. FaaSRank sustained the least number of inflight invocations during our experiments. Specifically, FaaSRank reduces the number of inflight function invocations by 59.62% and 70.43% freeing computational resources compared to the default Hashing scheduler on the Canada Cloud and AWS EC2 clusters, respectively. Function invocations scheduled by FaaSRank were completed faster overall than for the baseline schedulers.

Percentage of cold starts. We also recorded the percentage of cold-start function invocations in the experiments. Figure 6.4 shows the percentage of cold starts for individual schedulers on two OpenWhisk clusters. It's not surprising that the Default and Greedy schedulers achieved a higher number of warm-start executions, as they try to co-locate function invocations on invokers. This implies that FaaSRank trades off having a

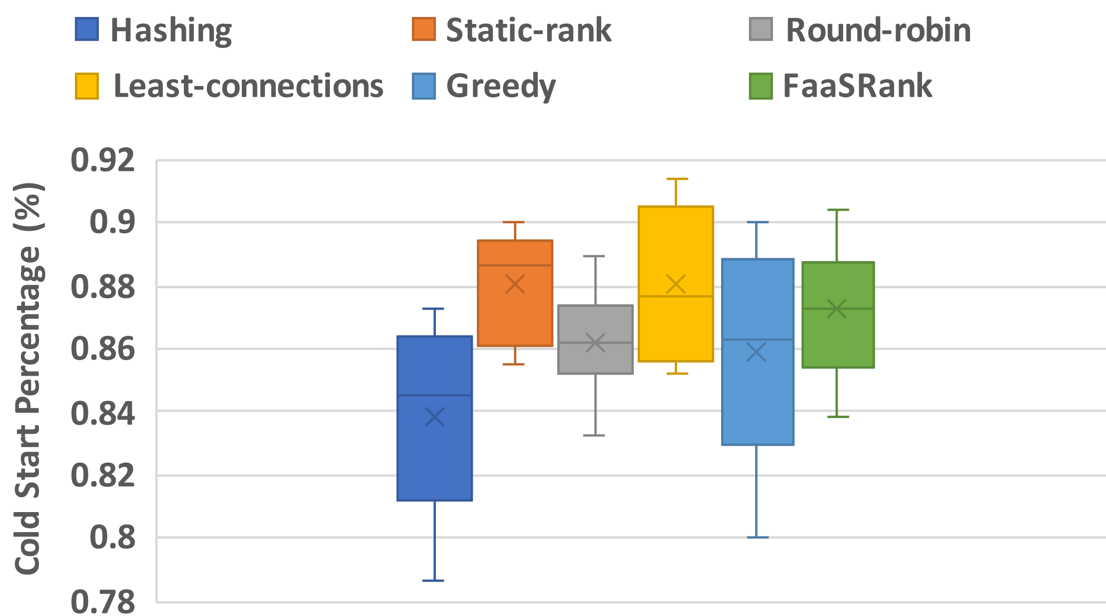


(a) Canada Cloud cluster

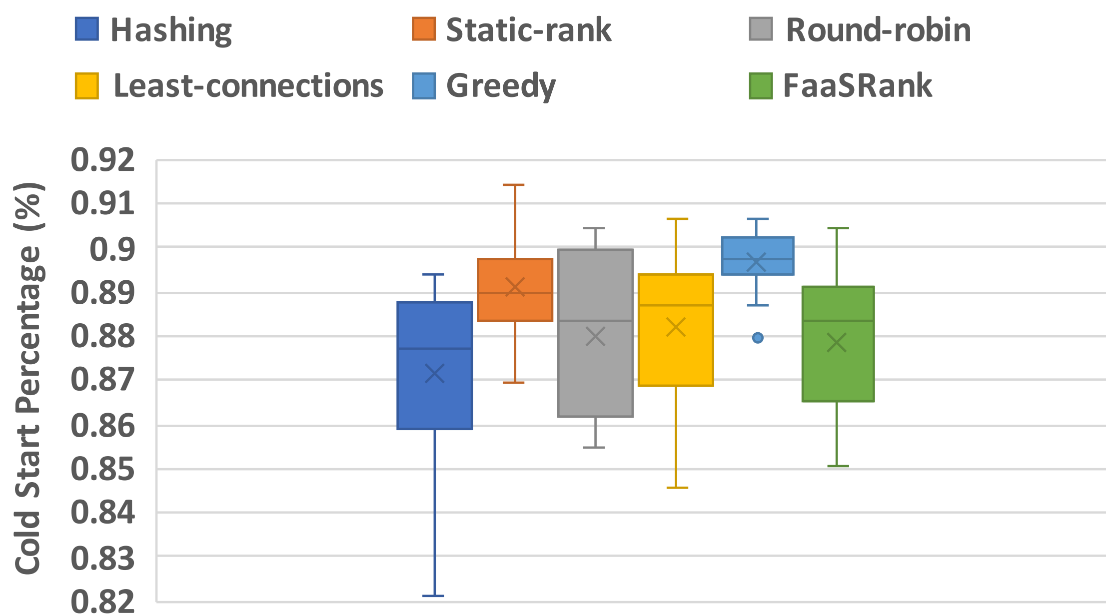


(b) AWS cluster

Figure 6.3: The time series of inflight invocations recorded on two OpenWhisk clusters



(a) Canada Cloud cluster



(b) AWS cluster

Figure 6.4: Percentage of cold starts for individual schedulers on two OpenWhisk clusters

higher number of warm starts with resource contention to achieve better overall scheduling decisions. FaaSRank's design that learns to optimize the overall average FCT of workloads enables the exploitation of this trade-off.

Compared to our baseline schedulers, FaaSRank was able to schedule function invocations for FaaS platforms by optimizing the objective, *i.e.*, overall average FCT of a workload. When scheduling, FaaSRank learns to trade off between invoker initialization and resource contention, a capability that traditional schedulers cannot easily provide. As FaaSRank reduces the overall average FCT of workloads, it provides excellent scheduling decisions for most of the functions while maintaining reasonable performance for the rest.

Chapter 7

RELATED WORK

In this chapter, we briefly review state-of-the-art schedulers and function scheduling approaches for FaaS platforms. We also discuss RL-based scheduling approaches in other contexts, and show the inefficiency of employing them directly in serverless FaaS platforms.

Schedulers for serverless computing. Prior research has investigated scheduling problems in serverless environments. Among all the FaaS schedulers, [31] has a most similar setting with FaaSRank. [31] leverages supervised learning to provide a function placement algorithm that schedules functions in serverless environments while incorporating resource utilization metrics of servers as input data while also trying to predict the performance of function placement. However, in contrast to our RL-based FaaSRank, one critical drawback of [31] is the requirement of independent profiling step before deployment, which must occur on a separated VM before the function can be used. This approach limits the immediate availability of the function deployment, which is counter to the goals of FaaS. FaaSRank gradually unearths training data as RL training proceeds, with no requirements of profiling phase before function deployment. Another limitation of [31] is the lack of a performance evaluation on a real serverless platform. We evaluate FaaSRank with extensive experiments using OpenWhisk clusters. [23] proposes a centralized scheduler for serverless platforms that assigns each CPU core of worker servers to CPU cores of scheduler servers for fine-grained core-to-core management. While the proposed approach seems promising, it doesn't consider resource conditions of worker servers when scheduling functions, and hasn't been evaluated in a real serverless platform. [47] presents a scheduler for managing resource in serverless platforms, which is implemented and evaluated on an OpenWhisk cluster. While [47] focuses on CPU power allocation, it only adopts a simple greedy algorithm to schedule functions. [30] uses RL to improve a container autoscaling service for Kubernetes clusters. In constrast, FaaSRank focuses on function scheduling as opposed to autoscaling

pods of containers. [19] presents a scheduler for optimizing distributed machine learning using a serverless computing service. In contrast, FaaSRank is designed to schedule any serverless workloads.

RL-based schedulers. Recently, many researches have applied RL techniques to various scheduling problems. [36] trains an RL agent to distribute TensorFlow computational graphs on devices. [8] proposes an RL-driven machine learning cluster scheduler that optimally places training jobs. [51] presents a hierarchical RL-based scheduler to schedule long-running applications in container clusters. [32] proposes an RL-based resource manager for a single machine environment, where FaaSRank is designed for distributed serverless clusters. [33] presents Decima, an RL-based scheduler to schedule DAG jobs for data processing clusters. Decima is implemented using a Spark framework by assigning executors to optimize average job completion time of workloads. FaaSRank optimizes a similar objective (overall average FCT) with Decima. However, Decima is not designed for serverless environments, as it ignores various features in serverless computing such as rapid changes in resource utilization, and warm vs. cold starts. Decima also doesn't consider the scheduling problem. Directly replacing FaaSRank with Decima in a serverless environment is not feasible. So far as we know, there are no prior solutions that optimize function scheduling in serverless platforms using RL.

Chapter 8
CONCLUSION

In this thesis, we present FaaSRank, an intelligent scheduler driven by deep reinforcement learning tailored to schedule functions in serverless platforms. FaaSRank employs neural networks to automatically learn function scheduling decisions. We implement FaaSRank in the Apache OpenWhisk platform, and evaluate FaaSRank against other baseline schedulers using realistic serverless functions and workload traces on two OpenWhisk clusters deployed on two public clouds: Compute Canada Cloud, and AWS EC2. Our results demonstrate that FaaSRank outperforms baseline schedulers by achieving the minimal overall average function completion time for our workloads while maintaining good individual functions performance. FaaSRank reduced the overall average function completion time by 9.25% and 10.10% respectively relative to OpenWhisk's default hashing scheduler. FaaSRank provided this improvement while reducing the average number of inflight function invocations per second by 59.62% and 70.43% for the duration of our workloads on the Compute Canada Cloud and AWS, effectively freeing computational resources compared to the default scheduler.

GLOSSARY

GRADIENT ASCENT: maximizing of the function so as to achieve better optimization used in reinforcement learning.

GRADIENT DESCENT: minimizing the cost function so as to find a local minimum used in machine learning.

TRAJECTORY: sequence of states and actions, which depicts the path of the agent through the state space up until the horizon.

ACTION SPACE: the set of possible actions that the agent can contemplate taking after observing the data.

EMBEDDING: a low-dimensional, learned continuous vector representation of discrete variables into which one can translate high-dimensional vectors.

POLICY NETWORK: neural networks as representation of policy, which dictates to the agent exactly what to do given the observed state.

ACTOR-CRITIC METHODS: methods where the actor takes as input the state and outputs the best action, and the critic evaluates the action by computing a value function.

DIFFERENTIABLE: the ability to compute the derivative of the operations in the module, and therefore to compute the gradients of the loss function with respect to the module parameters.

ADVANTAGE: the improvement compared to the average for an action taken at a given state.

BASELINE VALUE: the value subtracted from the total rewards to estimate the policy gradient with the aim of reducing variance.

REST: representational state transfer, a software architectural style which uses a subset of HTTP.

KEY-VALUE STORE: a data storage paradigm designed for storing, retrieving, and managing associative arrays.

CONTAINER: a virtual runtime environment that runs on top of a single operating system (OS) kernel that is designed to share an operating system rather than the underlying hardware.

BIBLIOGRAPHY

[1] Apache CouchDB. Apache CouchDB. http://couchdb.apache.org. [Online; accessed 1-May-2021].

[2] Apache Kafka. Apache Kafka. https://kafka.apache.org. [Online; accessed 1-May-2021].

[3] Apache OpenWhisk. Apache OpenWhisk Official Website. https://openwhisk.apache.org. [Online; accessed 1-May-2021].

[4] AWS. Amazon EC2 Dedicated Hosts. https://aws.amazon.com/ec2/dedicated-hosts/. [Online; accessed 1-May-2021].

[5] AWS. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/. [Online; accessed 1-May-2021].

[6] AWS. AWS EC2: Secure and Resizable Compute Capacity in the Cloud. https://aws.amazon.com/ec2/. [Online; accessed 1-May-2021].

[7] AWS. AWS Lambda: Serverless Compute. https://aws.amazon.com/lambda/. [Online; accessed 1-May-2021].

[8] Y. Bao, Y. Peng, and C. Wu. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *Proc. the IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[11] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *Proc. the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[12] Compute Canada. Compute Canada Cloud. https://www.computecanada.ca/. [Online; accessed 1-May-2021].

[13] Robert Cordingly, Wen Shu, and Wes Lloyd. Predicting Performance and Cost of Serverless Computing Functions with SAAF. In *Proc. the IEEE International Conference on Cloud and Big Data Computing (CBDCom)*, 2020.

[14] Google Cloud. Google Cloud Function:Event-Driven Serverless Compute Platform. https://cloud.google.com/functions. [Online; accessed 1-May-2021].

[15] WAND NETWORK RESEARCH GROUP. WITS: Waikato Internet Traffic Storage. https://wand.net.nz/wits/index.php, 2019.

[16] Urdaneta Guido, Pierre Guillaume, and van Steen Maarten. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH$_{c}omnet2009.html$.

[17] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. Fifer: Tackling Underutilization in the Serverless Era. *arXiv preprint arXiv:2008.12819*, 2020.

[18] Xinlei Han, Raymond Schooley, Delvin Mackenzie, O. David, and W. Lloyd. Characterizing Public Cloud Resource Contention to Support Virtual Machine Co-residency Prediction. In *Proc. the IEEE International Conference on Cloud Engineering (IC2E)*, 2020.

[19] Wang Hao, Niu Di, and Li Baochun. Distributed Machine Learning with a Serverless Architecture. In *Proc. the IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[20] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[21] Loshchilov Ilya and Hutter Frank. Decoupled Weight Decay Regularization. In *Proc. the IEEE International Conference on Learning Representations (ICLR)*, 2019.

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, 2019.

[23] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized Core-Granular Scheduling for Serverless Functions. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*, 2019.

[24] Vijay Konda and John Tsitsiklis. Actor-Critic Algorithms. In *Proc. the International Conference on Neural Information Processing Systems (NIPS)*, 2000.

[25] Kubernetes. Kubernetes Networking Services. https://kubernetes.io/docs/concepts/services-networking/service/. [Online; accessed 1-May-2021].

[26] W. Lloyd, S. Ramesh, Swetha Chinthalapati, Lan Ly, and S. Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *Proc. the IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[27] Wes Lloyd, Shrideep Pallickara, Olaf David, Mazdak Arabi, and Ken Rojas. Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services. In *Proc. the IEEE International Conference on Cloud Engineering (IC2E)*, 2017.

[28] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Performance Modeling to Support Multi-tier Application Deployment to Infrastructure-as-a-Service Clouds. In *Proc. the IEEE International Conference on Utility and Cloud Computing (UCC)*, 2012.

[29] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. In *Proc. the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[30] Schuler Lucia, Jamil Somaya, and Kühl Niklas. AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments, 2020.

[31] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. In *Proc. the 29th Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2019.

[32] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *Proc. the ACM Workshop on Hot Topics in Networks (HotNets)*, 2016.

[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proc. the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[34] Copik Marcin, Kwasniewski Grzegorz, Besta Maciej, Podstawski Michal, and Hoefler Torsten. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing, 2020.

[35] Microsoft Azure. Azure Functions. https://azure.microsoft.com/en-us/services/functions/s. [Online; accessed 1-May-2021].

[36] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. In *Proc. the International Conference on Machine Learning (ICML)*, 2017.

[37] NGINX. NGINX: High Performance Load Balancer, Web Server, Reverse Proxy. https://www.nginx.com/. [Online; accessed 1-May-2021].

[38] OpenAI. Gym: A Toolkit for Developing and Comparing Reinforcement Learning Algorithms. https://gym.openai.com. [Online; accessed 1-May-2021].

[39] OpenAI. OpenAI Spinning Up. https://spinningup.openai.com/en/latest/. [Online; accessed 1-May-2021].

[40] OpenFaaS. OpenFaaS Official Website. https://www.openfaas.com/. [Online; accessed 1-May-2021].

[41] PyTorch. PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. https://pytorch.org. [Online; accessed 1-May-2021].

[42] Redis. Redis Official Website. http://redis.io/. [Online; accessed 1-May-2021].

[43] NATIONAL LABORATORY FOR APPLIED NETWORK RESEARCH. WITS: Waikato Internet Traffic Storage. Anonymized access logs, 1995.

[44] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization, 2017.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[46] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. the USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[47] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *Proc. the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020.

[48] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proc. the International Conference on Neural Information Processing Systems (NIPS)*, 1999.

[49] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the Curtains of Serverless Platforms. In *Proc. the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.

[50] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[51] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[52] Y. K. Kim and M. R. HoseinyFarahabady and Y. C. Lee and A. Y. Zomaya. Automated Fine-Grained CPU Cap Control in Serverless Computing Platform. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2020.

[53] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing Serverless Platforms with ServerlessBench. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*, 2020.