This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

FaaSCtrl: A Comprehensive-Latency Controller for Serverless Platforms

Abhisek Panda, Smruti R. Sarangi

***Abstract*—Serverless computing systems have become very popular because of their natural advantages with respect to autoscaling, load balancing and fast distributed processing. As of today, almost all serverless systems define two QoS classes: best-effort (BE) and latency-sensitive (LS). Systems typically do not offer any latency or QoS guarantees for BE jobs and run them on a best-effort basis. In contrast, systems strive to minimize the processing time for LS jobs. This work proposes a precise definition for these job classes and argues that we need to consider a bouquet of performance metrics for serverless applications, not just a single one. We thus propose the comprehensive latency (CL) that comprises the mean, tail latency, median and standard deviation of a series of invocations for a given serverless function.**

Next, we design a system *FaaSCtrl*, whose main objective is to ensure that every component of the CL is within a prespecified limit for an LS application, and for BE applications, these components are minimized on a best-effort basis. Given the sheer complexity of the scheduling problem in a large multi-application setup, we use the method of surrogate functions in optimization theory to design a simpler optimization problem that relies on performance and fairness. We rigorously establish the relevance of these metrics through characterization studies. Instead of using standard approaches based on optimization theory, we use a much faster reinforcement learning (RL) based approach to tune the knobs that govern process scheduling in Linux, namely the real-time priority and the assigned number of cores. RL works well in this scenario because the benefit of a given optimization is probabilistic in nature, owing to the inherent complexity of the system. We show using rigorous experiments on a set of real-world workloads that *FaaSCtrl* achieves its objectives for both LS and BE applications and outperforms the state-of-the-art by 36.9% (for tail response latency) and 44.6% (for response latency's std. dev.) for LS applications.

***Index Terms*—job colocation, serverless computing, performance interference, reinforcement learning, resource scheduling.**

I. Introduction

The serverless computing paradigm is quickly becoming commonplace in almost all cloud and fog-based environments [1]–[5]. They are germane to such scenarios because of some of their beneficial features such as autoscaling, reduced operational costs, flexibility and fine-grained billing models [6], [7]. There are many celebrated examples of large corporations moving to a serverless paradigm such as Microsoft Azure Functions [8], Amazon Lambda [9], Google Cloud Functions [10] and IBM Cloud Functions [11]. As a result, there is a lot of research in industry and academia in this space [1], [12], [13]. Such frameworks are also increasingly being used in critical sectors like finance, healthcare, e-commerce and IoT [4], [7], [14]–[17].

Abhisek Panda is with the Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, email: abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi is with the Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, email: srsarangi@cse.iitd.ac.in

TABLE I: A summary of closely-related works to *FaaSCtrl*.

Work	Architecture	QoS classes	Evaluation metrics for the latency of the first class			
			Tail	Med.	Mean	Std. Dev.
Parties '19 [18]	MS	LC, BE	✓			
Clite '20 [19]	MS	LC, BE	✓			
LaSS '21 [7]	SL	LS			✓	
Sturgeon '22 [20]	MS	LS, BE	✓			
OLPart '23 [21]	MS	LC, BE	✓			
FaaSCtrl	SL	LS, BE	✓	✓	✓	✓

LC: Latency critical, BE: Best-effort, LS: Latency-sensitive, Med: Median and $Std.Dev.$: Standard Deviation, MS: Microservice, SL: Serverless

Platforms aim to improve the response latency to enhance user engagement and ensure timely completion of transactions in commercial applications such as e-commerce (Coco-Cola [22], Neiman Marcus [23]), finance (FINRA [24]), IT automation (Autodesk [24]) and video streaming (Netflix [25]). Similarly, for real-time applications, such as IoT backends (iRobot [26]) and real-time data analytics (Fannie Mae [24]), platforms aim to improve the response latency to enable real-time decision making. Even a minor variation in the response latency can adversely affect the aforementioned applications, causing a large overall slowdown and consequent revenue loss (refer to the cases described in [27]–[31]). Therefore, unlike many legacy frameworks, most studies on serverless technologies report a variety of performance metrics, not just the quintessential *mean response time* [1]. Table I shows a list of popular serverless and microservice-based intra-node resource managers, whose associated papers have been published in the last 5 years. It shows the latency metrics that they have considered, such as the mean, median, tail latency and standard deviation. We can see that they use a combination of metrics. This is mainly because serverless applications are very sensitive to noise and one metric does not convey the full picture.

In our experiments, we have observed that serverless applications are very sensitive to system noise, hence their execution time variation and mean latency are important parameters of interest. Sadly, optimizing one latency metric does not ensure that another latency metric will also improve. For instance, an application can experience an increase in the mean latency while meeting the target tail latency [32]. We

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

have observed many instances of this phenomenon in our experiments. We thus coined the new term comprehensive latency (CL) in this paper, which comprises a basket of latency metrics: mean, median, tail latency (99^{th} percentile) and standard deviation.

To ensure better comprehensive latencies for applications, commercial serverless platforms such as Microsoft Azure and Amazon Lambda provide two QoS classes: latency-sensitive (LS) and best-effort (BE). These platforms charge users differently for each QoS class [33], [34]. They, however, do not define the QoS guarantees that they provide very precisely. Platforms broadly aim to improve the throughput or average latency of BE applications only after meeting the requirements of LS applications (best-effort basis). It was thus incumbent upon us to provide a precise definition of these two QoS classes in terms of the comprehensive latency.

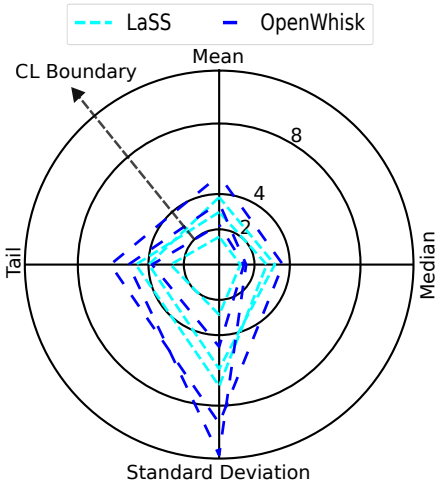


Fig. 1: The components of the comprehensive latency of LS applications, when executed with LaSS (a state-of-the-art serverless platform) and Apache OpenWhisk.

Figure 1 graphically shows our proposed definition. Let us consider a Kiviat plot that has four axes: mean, median, tail latency and standard deviation (four components of the CL). For a given LS application (function) let us consider all its invocations on a given node and calculate these metrics. For example, consider the mean. Let us compute the mean in a scenario with contention (several applications running) and a scenario where the application is run in isolation (no interference). Let us add a point in the Kiviat plot corresponding to the ratio $mean_{cont}/mean_{isol}$. Next, let us add points for the ratios of the rest of the three metrics as well and join them together to form a quadrilateral. Ideally, these four points should fall on the unit circle, which would indicate that contention had no effect. However, this is never going to be the case, hence, let us set a goal to bound all these four points within a circle with radius 2. The justification is that the response latency follows a lognormal distribution [35], [36] and the point 2 is close to the knee of the curve (also verified experimentally). We set this threshold for all the components of the CL. This definition gives us an absolute QoS guarantee for the CL subject to the system load being bounded (described in Section VIII-F). For BE applications, our aim is to minimize the radius of the circle that contains the corresponding quadrilateral.

In any real-world scenario, a platform schedules many different applications: LS and BE. A scheduling algorithm runs the risk of creating an *unfair* schedule that prioritizes one application over the other, or one metric over the other [37]–[40]. Given our definition of the CL and the fact that we are running many applications where applications can enter and exit the system at will, solving a complicated optimization problem within a few milliseconds is impossible. We need to simplify the scheduling problem. We use the method of surrogate functions [21], [41] in optimization theory to replace the master optimization problem with another simpler problem that yields a *good quality solution*. We perform a series of detailed characterization experiments to motivate the design of this function.

Our characterization experiments suggest that *resource contention* is the main reason for the degradation of the comprehensive latency of an application. Hence, understanding the causes of resource contention is very important. Based on our extensive analyses, we find that CPU wait time (time spent in a core's run queue), delays caused by non-voluntary context switches, cache contention, and locking are the main reasons for the degradation of the CL components in contemporary serverless applications. Our surrogate function was created using this information.

Once the reasons for CL degradation were understood, we looked at the main knobs that have been used to improve the comprehensive latency in prior work [14], [18], [19], [21]. These knobs are the assigned number of cores, allocated CPU time and memory bandwidth, and dedication of LLC (last-level cache) ways. We did not look at changing the memory bandwidth or interfering with cache managmennt because they were not found to scale with the number of processes. However, we found the first two knobs useful: the assigned number of cores and real-time process priorities.

Our scheme *FaaSCtrl* uses the aforementioned knobs to generate a schedule for LS and BE applications that is in line with our QoS definitions. It optimizes the surrogate function in an environment, where the behavior of tuning a given knob is not precisely known – the response function is uncertain. This means that all scheduling decisions need to be taken with partial information. This is a fit case for using reinforcement learning (RL), which has proved to be quite useful for similar problems in other domains [42]–[53]. We design a novel RL formulation that is on the lines of our surrogate function, and is intuitively explainable. We can compute a solution in milliseconds, which is an important requirement in a serverless setting. Finally, we validate the effectiveness of our design by evaluating *FaaSCtrl* under different colocation scenarios. Even though most of our evaluation is limited to a large multicore machine, we shall argue that it scales to clusters as well (based on some of our experiments).

To summarize, our contributions are as follows:

1) We identify the key sources of degradation in the comprehensive latency of an application when multiple serverless applications are colocated on a single host. Our key finding is that CPU contention, cache con-

tention, locking, and delays caused by non-voluntary context switches have a significant impact on the degradation of the comprehensive latency.
2) We define a surrogate function that guarantees the CL of LS applications and optimizes the CL of BE applications as per our novel definition of the terms LS and BE.
3) We design *FaaSCtrl*, which uses reinforcement learning to set the priority and the assigned number of cores. The RL algorithm does not allow applications to monopolize the CPU or compromise one metric for another.
4) When 5 LS applications and 5 BE applications are running simultaneously, *FaaSCtrl* improves the mean, median, tail latency and standard deviation of LS applications by 30.2%, 27.7%, 36.9% and 44.6%, respectively, with respect to the nearest competitor LaSS [7].

The rest of the paper is organized as follows. We discuss the relevant background for the paper in Section II, followed by a discussion of the related work in Section III. Subsequently, we profile microarchitectural counters and collect the execution statistics of an LS application when it is colocated with a BE application in Section IV. We motivate the need for an RL scheme to set the priority and the assigned number of cores of LS applications in Section V. We formally define the objective, metrics and constraints in Section VI. This is followed by explaining the design of *FaaSCtrl* in Section VII. Section VIII evaluates the efficacy of *FaaSCtrl* for reducing the CL. Finally, we conclude in Section IX.

II. Background

In this section, we provide the relevant background on serverless computing and reinforcement learning.

A. Serverless Computing

The serverless computing paradigm basically divides a large task into several smaller sub-tasks – each one is known as a serverless *function* that runs in an isolated environment. These functions can run on different network nodes in a distributed fashion. This idea is basically a second avatar of erstwhile web service based architectures; in this case, we can operate at the level of functions because of ultra-fast network speeds. Let us elaborate.

On a serverless computing platform, a developer only needs to write an application in the form of a *function chain*. In response to an external event, the platform executes each function of the function chain in a sandbox. To restrict a user's monopoly on resources, the user has to define resource requirements at the granularity of functions. In comparison to conventional cloud computing, such platforms provide auto-scaling features. In addition, it is easy to perform load balancing because we can just bring up more nodes that host the code of a certain frequently-accessed function. This paradigm is especially suitable for latency-sensitive (LS) applications like web applications, security applications, and IoT applications because functions have dedicated resources such as machines, and it is possible to optimize their runtime environment.

Such an environment has three major attributes: the number of idle sandboxes (*warm*) created for a function on the system; resource configuration, such as the number of sandboxes created for a function; and the degree of colocation of tasks that share resources.

1) Auto-scaling Features within a Single Node: A serverless platform spawns multiple sandbox processes for an application to meet its latency requirements. The number of sandbox processes associated with an application varies with the application's request arrival rate. In Apache Openwhisk [54], each of the sandboxes of an application is mapped to three multi-threaded processes: *containerd-shim, entrypoint script*, and *web server*.

2) Cold Start: Due to the on-demand execution of requests and associated security considerations, the platform spawns an ephemeral sandbox to serve a request. Therefore, the process of spawning a sandbox is on the critical path – this is referred to as *cold start*. To minimize the cold start latency, a serverless platform may use a light-weight sandbox mechanism [12], pre-warming, warm containers [55], [56], or checkpoint and restore-based techniques [13], [57], [58].

3) Multi-tenant Setup: In a typical multi-tenant setup, the serverless platform hosts multiple functions with varying arrival rates on the same machine; they share the limited compute and memory resources of the host among themselves. Therefore, colocation introduces interference in the function execution, and leads to slowdown and variation in the response latency [59], [60]. To limit the extent of slowdown and variation, the platform uses machine learning models to predict the performance of a function with colocation and place it on the least loaded node in a cluster [61].

B. Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique that employs an intelligent agent that takes actions in an uncertain environment, where the aim is to maximize the cumulative reward: the sum of the incentives received by the agent. The system is modeled as a Markov decision process (MDP) [62], which has a set of states ($\mathcal{S}$) that represent the environment, a set of actions ($\mathcal{P}$) that the agent can take, a function that denotes the probability of transitioning from state s to state s' for a given action a, and a reward function that provides the reward for the aforementioned state transition. The objective of any RL scheme is to come up with a policy π that determines the probability distribution of actions in a given state such that the expected value of the cumulative reward is maximized.

An RL technique needs a description of states, a list of actions, and a reward function. The policy π is computed automatically by the RL library; it basically uses a search technique. A common technique is the policy iteration technique [62] that relies on Monte Carlo simulation. Sadly, this technique is not suitable for a scenario like ours, where the variance in the rewards is high [62]. This is because we see a lot of contention and non-determinism in our system [63]. A newer set of approaches employs temporal difference methods [64], [65] that additionally associate a *value* with each state. It is in line with classical fixed-point approaches, where the state-value function represents the expected cumulative

reward that an agent will accumulate by starting from a given state. In this set, *FaaSCtrl* employs the actor-critic scheme because it has good convergence properties [66]. Again, within this subset, the advantage actor-critic (*A2C-RL*) method was found to be the best because of its superior results [67].

III. Related Work

A. Resource Scheduling in the Cloud

Prior work used resource partitioning techniques to meet the QoS guarantees for a mix of latency-critical and best-effort jobs on a node [18], [19], [68]–[71]. Parties [18] shifts resources from best-effort jobs to latency-critical jobs when the QoS is not satisfied and does the reverse when the QoS is easily satisfied. This approach cannot provide fairness and QoS simultaneously. To ensure both of these, Clite [19] and OLPart [21] use Bayesian optimization and RL techniques to find the optimal resource configuration. Sadly, these mechanisms do not take adequate measures to manage the CPU contention due to colocated applications (discussed in Section IV). In addition, the inherent design is intended for long-running applications, whereas serverless applications are typically short-lived (destroyed after execution).

B. Resource Scheduling in Serverless Computing

With the advent of latency-sensitive applications in serverless computing, the platform must provide bounded and predictable latencies for applications [72]. Prior work [7], [12], [13], [59], [63], [73] focused on improving the latency of applications but did not consider the performance variability in the latency in a multi-tenant setup. To improve the latency, prior work [7], [14], [74] focused on designing efficient resource scheduling schemes. They have highlighted that the degradation in the latency (w.r.t. native execution) is due to sharing limited resources on a single host machine [59].

Kaffes et al. [74] proposed a CPU core partitioning technique that assigns a dedicated CPU core to each request, thus minimizing resource contention. Nevertheless, when the load factor is high, allocating a dedicated core to each request has an effect on the waiting time of subsequent requests. Instead of using the core partitioning technique, Ensure [14] dynamically increases the CPU time of a sandbox process assigned to an application to mitigate contention. This scheme does not consider contention in other applications while taking an action – this has adverse implications on the overall fairness. In our work, we set the *resource configuration* of the container processes of an application to minimize variation in latency by capturing the complete state of the system (contention suffered by all the applications running on the system).

Instead of using heuristics, LaSS [7] uses a queuing theory-based model to determine the ideal number of container processes (that must be spawned for an application) to limit the waiting time. Furthermore, it maintains a few containers of an application in a standby mode to serve future requests. It does not explicitly consider the effects of colocation.

There is prior work that looks at cluster-level scheduling [73], [75]–[77], which is orthogonal to our work. We only look at scheduling decisions inside a physical machine. This can be coupled with efficient cluster-level scheduling schemes. The most important point to note is that prior work hasn't looked at ensuring determinism in terms of execution time, as we do.

C. RL-based Resource Scheduling

Grid computing [79], the Android OS [80] and cloud computing [41], [81]–[83] are just a few of the many fields that use RL-based schemes to schedule resources across applications in order to improve performance. In a cloud computing system, applications typically exhibit time-varying resource usage patterns. To improve the throughput and energy usage, Twig [82] and DRLPart [41] employ RL schemes to alter the allocated resources. They use the following features in their state representation: allocated resources to applications, number of threads, and HW counters related to caches, dTLB, branch instructions, and CPU clock cycles. To evaluate an action, they use the following metrics: throughput value, energy usage, and QoS. However, the reward function does not evaluate the impact of the resource decision on other running applications, thus lacking a complete view of the system in so far as fairness is concerned.

In contrast to prior work, *FaaSCtrl* uses the performance degradation that an application suffers, the overall system's fairness and the resource allocation of all the LS applications running on the system in the state representation. To capture the performance degradation, it adds microarchitectural counter values such as *#L1D_MPKI*, *#L2_MPKI*, and *IPC* to the state representation. The reward function explicitly includes the *fairness* metric (discussed in Section VII-C) to restrict LS applications from monopolizing resources.

IV. Characterization of a Serverless Framework

In this section, we use LaSS [7], a state-of-the-art serverless framework built on Apache Openwhisk [54], to analyze the slowdown and variation in the execution latency (time spent executing a request inside a container) when we run a latency-sensitive (LS) application alongside a best-effort (BE) application. Prior studies have shown that with colocation, the performance of applications degrades [18], [19]. Note that an advantage of pure serverless frameworks over competing frameworks like microservices is that it is very easy to dynamically allocate resources to a serverless function based on its execution characteristics, service time and arrival rate. Our experiments aim to determine the effect of the aforementioned parameters on the slowdown and variation in the execution latency.

A. Evaluation Methodology

We evaluate LaSS [7] on a physical server and study the CPU contention and the microarchitectural-level interference suffered by the container (sandbox) processes of an application. This is because we aim to design an intra-node resource scheduler that can be replicated across multiple nodes. The workloads comprise popular real-world serverless applications, which are summarized in Table II. We assign the QoS classes

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

TABLE II: Workloads used in the paper (adapted from highly cited prior work [14]–[16], [78]).

Application category	Workloads	Description	Service time (sec)	Max. arrival rate (RPS)	QoS class
Multimedia	Image Resizer (IR) [14], [78]	Resizes an image	0.003	650.0	BE
Web	Markdown Renderer (MR) [78]	Renders a markdown page as a HTML page.	0.125	90.0	LS
IoT	Object Detector (OD) [7]	Detects an object from a webcam feed using the SqueezeNet model.	0.222	8.0	LS
Web	Email Generator (EG) [14]	Sends an auto-generated email.	0.252	14.0	LS
Security	Binary Scanner (BS) [7]	Scans a binary for malware traces using YARA rules.	0.282	14.0	LS
Web	Stock Analyzer (SA) [14]	Analyzes stocks of a company for a given duration.	0.398	22.0	LS
Visualization	DNA Visualizer (DV) [15]	Visualizes DNA using the Python squiggle package.	0.493	12.0	BE
Scientific	Pagerank Checker (PC) [15]	Computes the page rank of a page in a graph based on the Barabasi-Albert model.	1.602	5.0	BE
Machine Learning	Product Review Analyzer (PRA) [14]	Generates a regression model using the logistic regression model trained on Amazon product reviews.	7.997	0.9	BE
Multimedia	Video Processor (VP) [15]	Converts a video into grayscale.	15.518	0.3	BE

TABLE III: System configuration

Hardware settings			
Processor	Intel Xeon Gold 6226R CPU, 2.90 GHz		
CPUs	1 Socket, 16 cores	DRAM	256 GB
Software settings			
Linux Kernel	5.14.9	Docker	v20.10.22

to the applications based on prior work [7], [59]. The system configuration is shown in Table III. We assume that the request arrival rate follows the Poisson distribution, and the service time is exponentially distributed (similar to LaSS [7]). To ensure a reasonable load on the framework, we determine the maximum request arrival rate based on the system's configuration and capacity. Similar to Parties [18] and Twig [82], we run a serverless application with an increasing arrival rate until the tail latency increases exponentially. In this experiment, we pin the container processes to all system cores and set the scheduling policy to the standard Linux time-sharing scheduler policy (*CFS*).

To measure the variation in the execution latency, we employ the following statistical measures: standard deviation, and mean. In our study, we compare the value of the aforementioned metrics with respect to when an application is executed in isolation.

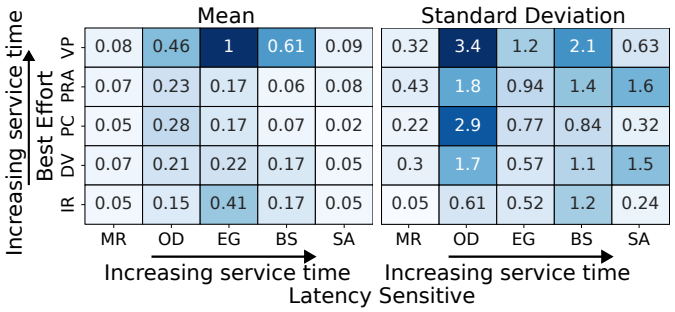


Fig. 2: Sensitivity of an LS application's execution delay when co-located with a BE application, both of which are running under moderate load. Each cell indicates an increase in the std. dev. or mean value of an LS application's execution latency relative to isolated execution.

B. Quantifying the Interference

To capture the effect of CPU contention and the microarchitectural-level interference suffered by an LS application, we measure the mean and std. dev. of the execution latency. For ease of analysis, we execute each of the LS applications along with each of the BE applications with a moderate fraction of their maximum arrival rate (usually between 40% and 60%) [18], [82]. In Figure 2, we show that the mean and std. dev. of the execution latency of an LS application increase by up to 1× and 3.4×, respectively, when colocated in contrast to when executed in isolation. *Note that each of the subfigures (part of Figure 2) contains ratios*.

When an LS application is colocated with the video processor (VP) application, the std. dev. and mean execution latency degrade the most in comparison to other BE applications. This degradation is because the VP application has a longer service time (= 15.5 seconds) than other BE applications. The degradation in the execution latency of LS applications with a high arrival rate (markdown renderer) and a high service time (stock analyzer) is relatively lower compared to other LS applications. This is because the number of containers serving the applications is high (at least 7).

1) The mean and std. dev. of the execution latency of an LS application can increase by up to 1× and 3.4×, when executed alongside a BE application with a high service time (=15.5 seconds).
2) If the number of containers serving an LS application is high (at least 7) as a result of the application's arrival rate or service time, we find minimal performance degradation.

C. Understanding the Interference

We collect hardware performance events (HPEs) provided by Intel Xeon processors and CPU-usage events from the *proc* file system during the execution of an application's request. The CPU-usage events capture the per-process CPU wait time (time spent by a process and its threads waiting in the run queue of a core) and the total number of non-voluntary context switch events (*#nvcs*). Using Pearson's correlation coefficients, we next perform statistical analyses to determine

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

which HPEs and CPU-usage events are correlated with the execution latency of LS applications.

TABLE IV: Pearson's correlation coefficients of HPEs and CPU-usage events with the execution latency of LS applications.

Events	MR	SA	EG	BS	OD
L1DMPKI	0.50	0.51	0.37	0.23	0.10
L1IMPKI	0.38	0.49	0.35	0.25	0.31
L2MPKI	0.42	0.52	0.20	0.26	0.02
LLCMPKI	0.30	0.10	0.26	0.45	0.07
iTLBMPKI	0.11	0.23	0.16	0.30	0.08
dTLBMPKI	0.28	0.11	0.15	0.19	0.37
IPC	0.69	0.58	0.43	0.70	0.23
CPU wait time	0.79	0.68	0.63	0.69	0.80
#nvcs	0.42	0.43	0.29	0.55	0.59

MPKI: Misses per kilo-instructions

In Table IV, we list the correlation of HPEs and CPU-usage events with the execution time of an application. Among the HPEs, L1D MPKI (*#L1D_MPKI*), L2 MPKI (*#L2_MPKI*) and *IPC* are correlated with an LS application's execution latency. This is a direct outcome of destructive interference caused by colocation. Furthermore, it is consistent with prior studies on resource interference with colocation [18], [19]. Among the CPU-usage events, the CPU wait time and the *#nvcs* events are highly correlated with an LS application's execution latency. This observation is specific to the serverless computing paradigm. Let us elaborate.

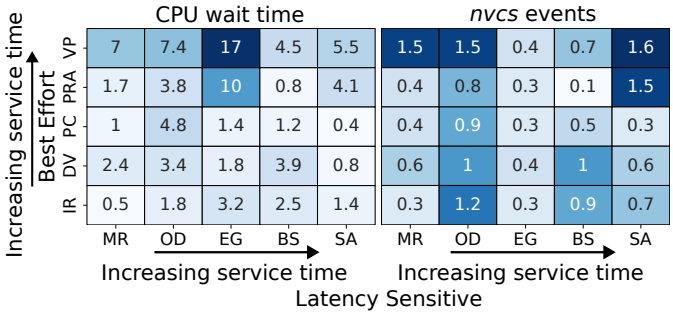


Fig. 3: Normalized CPU-usage counter values of the LS application (with respect to when executed in isolation)

1) Analyzing the CPU Contention: In Figure 3, we show that when an LS application is colocated with a BE application, the CPU wait time and the *#nvcs* events increase by up to 17× and 1.6×, respectively. This is because the operating system treats container processes as regular processes and employs the standard time-sharing (SCHED_OTHER) scheduling policy. Thus, the scheduler fairly distributes the CPU time between containers of LS and BE applications. We observe that the number of CPU-clock cycles required to execute a request of an LS application ranges between 97 and 1715 cycles. Consequently, the time spent by the container processes in the run queue for execution has a significant impact on the execution latency, resulting in execution time variation. Note that a container is mapped to three multi-threaded processes: *containerd-shim, entrypoint script*, and *web server*.

We observe a higher degree of CPU contention in the object detector application due to the file-level futex lock on *libcaffe2.so* of the py-torch library that is shared across the containers running on different cores.

If an LS application such as OD employs a locking mechanism, then the slowdown and variation in the execution latency can be partially accounted for by the application's container processes waiting to acquire the lock.

V. Motivation

In this section, we motivate the need for a framework that intelligently regulates the allocated CPU resources to minimize the slowdown and variation in the latency of latency-sensitive (LS) applications. Furthermore, we must ensure that the resource decisions made by the framework are independent of user-supplied input parameters. Note that we use the same experimental setup discussed in the previous section.

A. Default Knob: Assigning Priority

To minimize the CPU contention, the following scheduling configurations based on increasing the priority can be applied to the container processes of LS applications: ① increase the priority (nice) value according to the SCHED_OTHER policy (time-sharing scheduling policy) (*TS*), or ② change the scheduling policy from SCHED_OTHER to SCHED_RR (round robin real-time scheduling policy) (*RT*). We found SCHED_FIFO to be quite sub-optimal. In the case of the TS configuration, the majority of the processor time is allocated to LS containers. While in the RT configuration, the OS prioritizes LS containers over other programs running with the SCHED_OTHER policy (the time slice is 100 ms [84]).

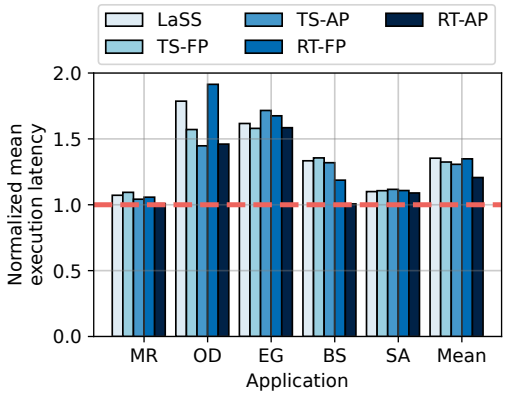


Fig. 4: Sensitivity of the execution latency of LS applications running with a priority assignment scheme when colocated with all BE applications. *Note that the RT-AP priority allocation scheme performs marginally superior than other schemes.*

Let us discuss the efficacy of the TS and RT configurations on all LS applications when colocated with all BE applications. For this study, we use the following priority assignment schemes for LS container processes in both of the scheduling configurations: ① fixed priority (*FP*), and ② arbitrary priority (*AP*). SCHED_OTHER priority values range from 100 to 139 (lowest priority), whereas SCHED_RR priority values range from 1 to 99 (highest priority). Under the fixed priority

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

scheme, *TS-FP* assigns the priority value 100, while *RT-FP* assigns the priority value 80 [85]. With the arbitrary priority scheme, we assign unique priority values in a round robin fashion to each LS application. *TS-AP* assigns priority values within the range of 100 to 129, whereas *RT-AP* assigns priority values within the range of 1 to 80. In the case of the *TS-AP* scheme, we choose the priority values with a step size of 4. As a result, no two applications receive roughly the same time slice. **Note that for SCHED_RR, scheduling priorities, higher is not always better [85].**

In Figure 4, we plot the mean execution latency for LaSS and four of the priority assignment schemes. All of them are normalized to the isolated execution case. We run all the LS and BE applications with a moderate fraction of their maximum arrival rate (between 5% and 15%). We observe that the RT-AP scheme improves the mean execution latency by 10.8%. Even though the RT-AP scheme is only marginally superior, it is possible to devise an efficient priority assignment scheme that dynamically alters an application's priority.

Note that the RT-FP scheme is not always the best. The reason for this is that in modern serverless applications, there is a very complex interaction between the container processes, the framework, and the OS. As a result, increasing the priority of some processes hurts the application in the long run because kernel threads and daemons that run with the real-time priority 50 do not run that frequently (also observed in [86], [87]). Furthermore, setting a fixed priority introduces contention between the container processes of LS applications.

The SCHED_RR scheduling policy with an arbitrary priority scheme allocates more CPU time to LS applications and mitigates the CPU contention between LS applications, but setting of unique priorities to applications is non-trivial.

B. Setting the Physical CPU Affinity

To minimize the cache miss rates we can restrict the container processes of an LS application to a set of physical cores. For this study, let us partition the available physical CPU cores in an $m : n$ ratio, where m represents the number of physical cores allocated to the LS application and n represents the number of physical cores allocated to the colocated BE application. VP was chosen as the BE application because of its high service time. We execute each of the LS applications along with VP with a moderate fraction of their maximum arrival rate (usually between 40% and 60%) [18], [82].

In Figure 5, we show that the mean execution latency of the markdown renderer (MR) and stock analyzer (SA) applications improves with a larger number of dedicated physical cores. For the object detector (OD) application, the degradation is quite large when the physical CPU affinity is set, compared to LaSS [7]. This is because OD employs futex locks on a shared library that is shared with all the containers, thus increasing the voluntary context switch events by up to 3.6×. For the rest, the results are subpar. As a result, a more intelligent mechanism is required to set the physical CPU core allocation of an LS application.

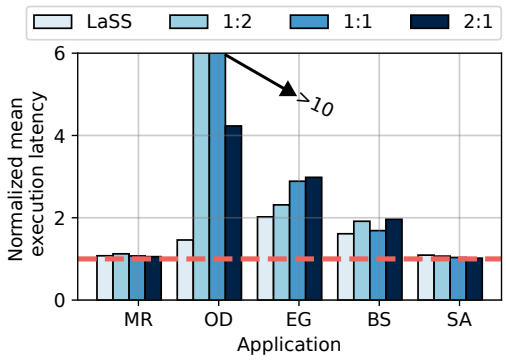


Fig. 5: Sensitivity of the execution latency of LS applications with varying ratios of ⟨#LS cores⟩:⟨#BE cores⟩ when colocated with the VP application, which is a BE application. *Note that the performance of OD is worst with core allocation.*

Setting core affinities does not help all the time, regardless of the $m : n$ ratio. It is particularly pernicious for applications like OD that access futex locks. A more intelligent scheme is needed.

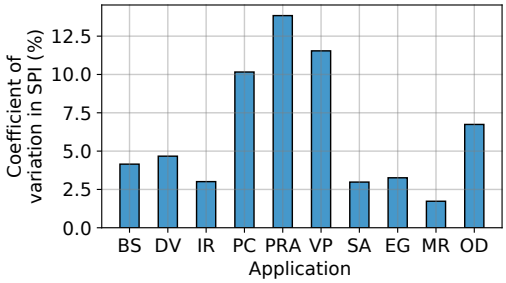


Fig. 6: The σ/μ values (coefficient of variation) of the SPI metric of applications as the input size changes.

C. Effect of the Input Size

In a serverless computing platform, an external user provides the input. To restrict the resource scheduling decision's sensitivity to input size, the framework must employ a metric that varies minimally with the input size of the application. The latency related metrics are sensitive to the input size. Therefore, we measure the seconds-per-instruction (SPI) metric of an application by varying the size of the input. The SPI quantifies the instruction execution rate of a function. To do this experiment, we considered larger inputs (up to 5× more size). For lack of space, we are not listing all the sources of the new inputs, however, the same can be provided on request. For instance, for Image Resizer (IR), we just downloaded random images from Google Images of the appropriate size, or for Video Processor (VP), we downloaded a random video from YouTube. The choice did not make a difference. As we can see in Figure 6, the mean SPI across the inputs remained stable (σ/μ within 5.05% on an average).

VI. Problem Formulation

In this section, we formulate the resource allocation problem for a mix of serverless applications belonging to two QoS

classes: LS and BE. Since serverless platforms are multi-tenant setups, it is crucial to maintain fairness [38]–[40]. The objective is to ensure that all the components of the comprehensive latency for an LS application are limited to the CL boundary $\langle 2, 2, 2, 2\rangle$ without negatively impacting the latency of other LS and BE applications. To evaluate whether a resource configuration meets the objective or not, we need to define suitable metrics and their respective constraints.

Prior work has used different latency metrics or instructions per cycle (IPC) of an application to determine whether a resource configuration met the objective or not. However, Haque et. al. [32] and our experiments have demonstrated that optimizing one latency metric does not ensure that another latency metric will also improve. For instance, an application can experience an increase in the mean latency while meeting the target tail latency [32]. This approach contradicts our objective of enhancing all aspects of the comprehensive latency. Furthermore, the input's size influences the comprehensive latency components of an application. As a result, *FaaSCtrl* relies on the performance metric (S^{perf}) and the fairness metric (S^{fair}) to evaluate an LS application's resource configuration – these are relatively independent of the input size and the number of instructions executed by the function.

We use these metrics to design a surrogate function, a classical optimization technique where the surrogate function approximates the objective function (similar to OLPart [21] and DRLPart [41]). Let us elaborate.

$$\begin{aligned} SPI &= \frac{execution\ time}{\#insts} \\ &= \left(\frac{\#insts}{freq * IPC} + \textit{CPU wait time}\right) \Big/ \#insts \\ &= \frac{1}{freq * IPC} + \frac{\textit{CPU wait time}}{\#insts} \end{aligned} \tag{1}$$

$$S^{perf} = \frac{SPI^{alone}}{SPI^{shared}} \tag{2}$$

1) Performance (S^{perf}): The S^{perf} metric captures the performance degradation of an LS application in the previous time epoch. We quantify performance in terms of the seconds per instruction (SPI) metric, which denotes the reciprocal of the instruction execution rate (a higher SPI value denotes greater performance degradation). We first compute the SPI metric of the application using Equation 1, where $freq$ is the CPU clock frequency of the machine. Subsequently, we define the application's performance degradation as the ratio of the SPI metric when running in isolation (SPI^{alone}) to that when running with other applcations (SPI^{shared}) (see Equation 2). The former is provided by the developer. Ideally, it should be 1 (contention has not effect). However, in practice it will always be less than 1. The stability of the SPI metric of a serverless application with a change in the input size was discussed in Section V-C.

Constraint: In terms of performance, our objective is to ensure that all the components of the comprehensive latency for all LS applications are within the CL boundary. In order to comply with this objective, a resource configuration must ensure the value of S^{perf} exceeds 0.5 for all LS applications. Note that this is a tighter condition than the CL boundary constraint. Enforcing this enforces the CL boundary constraint.

$$S^{fair} = \frac{min_i S_i^{perf}}{max_j S_j^{perf}} \tag{3}$$

2) Fairness (S^{fair}): When applying a resource configuration, it is important to consider both LS and BE applications without sacrificing the latter's performance to improve the former [38]–[40]. To capture fairness, we formulate the S^{fair} metric (conceptually similar to [88]). Subsequently, we define the fairness metric S^{fair} as the ratio of the minimum performance degradation to the maximum performance degradation across all applications running on the system (see Equation 3) (ideally it should be 1).

Constraint: In terms of fairness, our objective is to ensure that an LS application's resource configuration does not steal resources from other LS and BE applications. To conform with this, we must ensure that the S^{fair} metric exceeds τ.

3) Objective Function: Ideally, we need to solve a multi-objective optimization problem where the S^{perf} of every application is maximized. However, that is impractical – there are too many objectives. We can instead try to maximize the S^{perf} of all LS applications. Such multi-objective optimizations problems are hard to solve because the solutions are mostly on the Pareto-optimal front and thus it becomes necessary to prioritize one LS application over the other. We shall define a simpler version in Section VII-C3, which looks at the optimization function from the point of view of a single LS application (before its scheduling decision is made). Any such objective function needs to be a combination of the S^{perf} of that application and the system fairness. We shall elaborate further in Section VII-C.

VII. Design

In this section, we discuss the design of *FaaSCtrl* that sets the resource configurations of applications to ensure that all the components of the comprehensive latency for LS applications are limited to the CL boundary without negatively impacting the latency of BE applications. In Section V, we showed that simple priority assignment and physical CPU core affinity setting schemes do not provide good results all the time – there is no clear winner. The performance of these methods varies depending on the workload. *FaaSCtrl* must consider the dynamic resource contention of each application across all possible colocation scenarios. An exhaustive search technique to find the best resource configuration is expensive and time consuming, because the number of possible resource configurations is an exponential function of the number of colocated LS applications. Furthermore, we need to profile the system under different colocation scenarios to design an analytical model. Since the primary task is decision making in an uncertain environment, reinforcement learning (RL) based techniques are naturally germane to such scenarios (similar to prior work that targets conceptually similar problems [41],

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

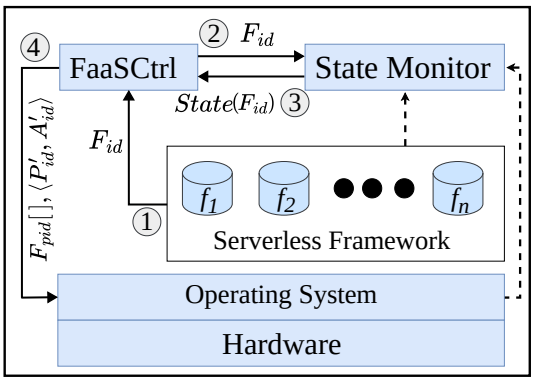


Fig. 7: The high-level design of *FaaSCtrl*.

TABLE V: State attributes of a serverless application (F_{id}).

Notations	
P_{id}	The single priority allocated to the container processes.
C_{id}	The single physical CPU core allocation of the container processes. Represents the no. of CPU cores allocated.
F_{lock}	A Boolean variable denoting whether an application uses futex locks that is shared across containers.
F_{slow}	A floating point (FP) value denoting the ratio of the IPC metric of the serverless function to that when executed in isolation.
F_{wait}	A FP value denoting the ratio of the CPU wait time to the number of instructions.
Cnt_{id}	The number of containers serving the serverless function F_{id}.
$X_{events}[]$	A vector representing the *#nvcs* events, *#L1D_MPKI* and *#L2_MPKI*.
S^{fair}	A FP value denoting the overall system's fairness (see Section VI-2)
P_{low}	The number of container processes that use the SCHED_RR scheduling policy and their priority is $\leq P_{id}$.
P_{high}	The number of container processes that use the SCHED_RR scheduling policy and their priority is $> P_{id}$.
C_{other}	The total number of cores currently allocated to other LS applications running on the system.

[80]–[82]). Therefore, *FaaSCtrl* uses the Advantage Actor-Critic reinforcement learning (*A2C-RL*) methodology [89] to decide the resource configuration of an application

A. High-Level Overview

In Figure 7, we show the high-level design of *FaaSCtrl*. The serverless framework sends a request to *FaaSCtrl* in order to set the *resource configuration* of a serverless function (F_{id}) (indicated as ① in Figure 7). Subsequently, *FaaSCtrl* fetches the *state* of the application and the entire system $State(F_{id})$ from the state monitor daemon (② and ③). The fields of the state are shown in Table V. Note that defining the state is a very tricky process in any RL scheme. We have opted for a partially observable strategy, where the state from the point of view of an application comprises some of its execution parameters/attributes and an estimate of the behavior of the rest of the system. The fields P_{id}, C_{id}, F_{lock}, F_{slow}, F_{wait}, Cnt_{id}, and $X_{events}[]$ in Table V are application specific. The other terms represent the approximate behavior of the rest of the system. They will be elaborated in the subsequent sections. Note that we use the $[]$ symbol to indicate vectors.

After receiving the state, *FaaSCtrl* computes the best resource configuration for an LS application's containers such that the constraints are met using the *A2C-RL* methodology. The resource configuration is a 2-tuple of the new priority P'_{id} and a new physical CPU allocation C'_{id} for the containers of the application (④). Subsequently, *FaaSCtrl* utilizes the *chrt* and *taskset* utilities in Linux to set the priority of the container processes and the core affinities of an LS application, respectively [90], [91].

B. State Monitor

The state monitor daemon is responsible for collecting the *state* of an application+system and providing it to *FaaSCtrl*. The attributes of the state are shown in Table V. The state captures the following information about an application: events influencing the execution latency, resource allocation and lock usage (F_{lock}). In addition, it also captures the overall system's fairness (S^{fair}) and resource allocation of other LS applications (P_{low}, P_{high} and C_{other}). To identify degradation in the execution latency, the state collects the following events: *L1D MPKI*, *L2 MPKI*, *#nvcs*, F_{wait} and F_{slow} (discussed in Section IV and Section VI). Similarly, it uses P_{id}, C_{id}, and Cnt_{id} to capture the resource allocation of the application. It also uses the *perf* tool to collect the `syscalls:sys_enter_futex` event, which shows if a serverless application uses futex locks shared between containers (F_{lock}) (discussed in Section V-B). Finally, it computes the S^{fair} metric using Equation 3 to capture the fairness among LS+BE applications.

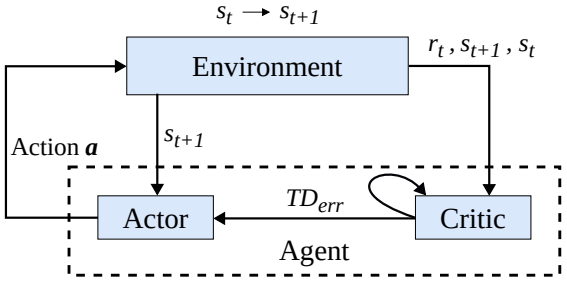


Fig. 8: The working of *FaaSCtrl*

C. FaaSCtrl

FaaSCtrl computes the best resource configuration of LS applications that conforms to the constraints on the S^{perf} and S^{fair} metrics using the A2C-RL model. It sets the real-time priority and the number of assigned cores for LS applications to ensure that they meet the CL condition. For BE applications, we use the default user-level CFS scheduler and pin the container processes to all system cores. Regardless of their *nice* value, their priority will always be lower than that of an LS real-time process. Additionally, given that we will have many LS applications, setting the core affinity of BE applications did not prove to be very beneficial. We design an appropriate reward function that maximize the overall fairness, thereby ensuring that LS applications will not overly penalize the BE applications. Let us explain with an example. Assume that an LS application gets prioritized and its S^{perf}

increases to 0.75 from 0.5 (minimum). This may affect the overall fairness and cause it to reduce. Given that maximizing fairness is one of our sub-objectives, the system will not allow the fairness to reduce. It will either stop the RL algorithm from improving the S^{perf} of the LS application given that it is already within the CL boundary, or it will improve the performance of BE applications such that the fairness improves. In both the cases, BE applications' performance improves and they are not discriminated against. The RL algorithm pulls them forward (improves their performance) owing to the fairness metric and the fact that it is tied with the best-performing LS application. Let us discuss in detail.

1) Overview: In Figure 8, we show the working of *FaaSCtrl*. The environment represents the system, including the serverless framework, state monitor daemon, system hardware, and operating system. For each time epoch t, the environment sends applications' features to *FaaSCtrl* through the state monitor daemon. The agent extracts the LS application's current state $\mathbf{s_t}$. Subsequently, the agent applies action a_t to the environment, expecting a higher reward (an action is referred to as a *scheduling action* in this paper). In the next epoch $t+1$, the environment's state changes to $\mathbf{s_{t+1}}$. The agent then receives a reward r_t from the environment for action a_t. The agent then uses this reward to train the actor and the critic artificial neural networks to improve their decision-making capabilities.

2) Scheduling Actions: For any given state $\mathbf{s_t}$ of an LS application, we aim to bound the comprehensive latency within the CL boundary, while ensuring fairness in the system. Our characterization study revealed that CPU contention and cache contention are the primary sources of performance degradation in an application's execution latency. In Section V, we have shown that using the SCHED_RR scheduling policy reduces an LS application's CPU contention. Furthermore, we demonstrate that smartly assigning the number of CPU cores while maintaining core affinity improves the miss rates. When assigning CPU cores, we ensure that the set of cores assigned to the task remains more or less the same over consecutive epochs. As a result, we are able to maintain the CPU core affinity of applications. With this insight, we design a set of scheduling actions that alter the real-time priority and the assigned number of cores for LS applications by a fixed step size (conceptually similar to Parties [18]). Each of the scheduling actions is represented in the form of a $\langle \Delta P, \Delta C \rangle$ tuple, where $\Delta P \in \{-P_{step}, 0, P_{step}\}$ and $\Delta C \in \{-\infty, -C_{step}, 0, C_{step}\}$. If the value of ΔC is $-\infty$, then we revoke the physical CPU core allocation. The size of the scheduling action space is 12.

$$R = \begin{cases} a\,S^{fair} + b\,S^{perf} & \text{iff } S^{fair} > \tau \text{ and} \\ & max_{i \in LS}\ (S_i^{perf}) \geq 0.5 \\ -1 & otherwise \end{cases} \quad (4)$$

$where,$ a, and b are positive constants.

3) Reward Function: To direct *FaaSCtrl* effectively within the extensive search space, we create a reward function to assess the quality of an LS application's resource configuration (refer to Equation 4). We categorize the system's overall state into two classes: desired and undesired. The *desired* state indicates that all LS applications comply with the CL boundary while maintaining overall system fairness ($> \tau$). Conversely, the *undesired* state signifies that either the comprehensive latency of LS applications is beyond the CL boundary, the system is unfair or both. If the state falls into the undesired class, we assign a negative reward (-1), indicating that the RL model should avoid this action in the future [21]. Once the state is in the desired class, we need to solve a multi-objective optimization function that maximizes both S^{perf} (for each LS application) and S^{fair}. We convert this to a single-objective optimization function while scheduling an LS application by alternatively defining the objective/reward as $a \times S^{fair} + b \times S^{perf}$. Note that S^{perf} is application-specific whereas S^{fair} holds for the full system.

4) Agent: The agent is responsible for finding a good scheduling action for a given state of the system and applying it to the environment. It consists of two artificial neural networks (ANNs): actor and critic. Actor finds a scheduling action for a given state. Since the scheduling action space is discrete, we use the Softmax function at the output layer of the ANN. Conversely, critic estimates the value function ($V(s)$), which predicts the expected return from a given state. After applying the scheduling action a_t to the environment, the environment sends the previous state $\mathbf{s_t}$, current state $\mathbf{s_{t+1}}$, and reward r_t to the critic network and the current state $\mathbf{s_{t+1}}$ to the agent network. Subsequently, critic calculates the temporal difference error using Equation 5. On every training step, we update the critic ANN using the mean squared error loss function and the actor ANN using the policy gradient method [92].

$$TD_{err} = r_t + \gamma\, V(\mathbf{s_{t+1}}) - V(\mathbf{s_t}) \quad (5)$$

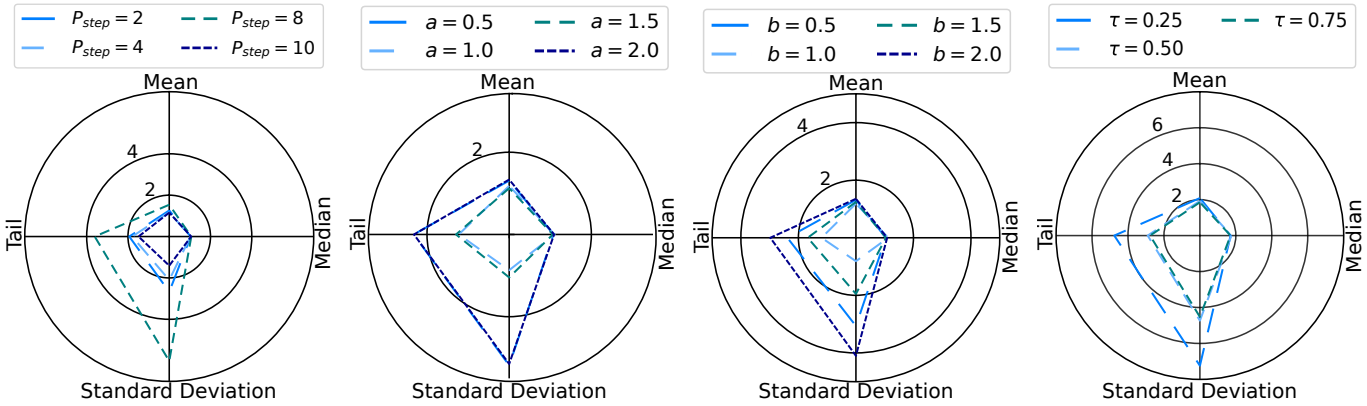
Here, γ ($\in [0, 1)$) is an ageing factor. γ is set to 0.99 (on the lines of [93], [94]).

5) Learning the Optimal Policy: To improve the learning of the ANNs, we employ the ϵ-greedy algorithm [95] during the learning phase. In the ϵ-greedy algorithm, the agent selects a random action (exploration) over the best possible action (till known) for a given state with a probability of ϵ. We set ϵ= 0.1 in our system [92]. In addition, to save computation time during training of the ANNs [96], we eliminate the policies where Δ C $\in \{-C_{step}, 0, C_{step}\}$ if the F_{lock} attribute is set (= 1) in the state of an application. This is because applications employing futex locks shared among containers should not have dedicated CPU cores (as discussed in Section V-B). We have used a policy where we do not have offline learning (exploration) and online inferencing (exploitation). For us, both phases are interleaved and both are online. There is no profiling phase. We start with an exploration:exploitation ratio of 5:1 and then linearly decrease this ratio to 1:100. Then we run workload combinations that haven't been seen in this time frame – clean separation between train and test.

VIII. Evaluation

In this section, we evaluate the components of the comprehensive latency of applications using *FaaSCtrl* and compare

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015



(a) CL of LS applications $\langle a = 1, b = 1, \tau = 0.5 \rangle$

(b) CL of LS applications $\langle P_{step} = 10, b = 1, \tau = 0.5 \rangle$

(c) CL of LS applications $\langle P_{step} = 10, a = 1, \tau = 0.5 \rangle$

(d) CL of BE applications $\langle P_{step} = 10, a = 1, b = 1 \rangle$

Fig. 9: The average comprehensive latency of LS and BE applications when 5 random pairs of LS and BE applications are executed on a machine by changing the values of the hyperparameters: P_{step}, $a(S^{fair})$, $b(S^{perf})$ and τ using the one-at-a-time hyperparameter tuning method.

it against three state-of-the-art proposals: LaSS [7], Clite [19] and OLPart [21]. Recall that we had already discussed the evaluation setup and benchmarks in Section IV. We implement these designs on a popular open source serverless framework, *Apache OpenWhisk v1.0* [54].

TABLE VI: Hyperparameters in the *FaaSCtrl* design.

Hyperparameters		
τ	The minimum allowed value of the fairness metric (S^{fair}).	0.5
P_{step}	A scalar value representing the change in the priority of an application.	10
C_{step}	A scalar value representing the change in the physical CPU allocation of an application.	1
a	The weight of the S^{fair} metric in the reward function.	1
b	The weight of the S^{perf} metric in the reward function.	1

We first discuss the configuration of the actor and critic artificial neural networks of *FaaSCtrl* in Section VIII-A. To fine-tune hyperparameters and ensure convergence of the neural networks, we perform a sensitivity analysis in Section VIII-B. In Table VI, we show the values of the hyperparameters that we use in all of our experiments. To analyze the impact of *FaaSCtrl* on LS and BE applications in detail, we evaluate *FaaSCtrl* for all possible combinations of a single LS and a single BE application first (see Section VIII-C). In Section VIII-D, we evaluate *FaaSCtrl* on a full execution (5 LS + 5 BE applications). Additionally, we evaluate the influence of the number of BE applications on LS applications in Section VIII-E. We further experimentally determine the load up to which *FaaSCtrl* can maintain fairness and ensure the performance constraints in Section VIII-F.

A. Configuration

The actor and critic ANNs of *FaaSCtrl* have two layers. The first layer of actor and critic contains 200 neurons with ReLU serving as the activation function [92]. The second layer of Actor contains 12 neurons, with Softmax as the activation function that represents the probability distribution of the scheduling action space. The second layer of Critic contains 1 neuron that represents the expected return from a given state. We set the learning rate to 0.0001 (on the lines of [93], [94]).

B. Analyzing Hyperparameter Sensitivity

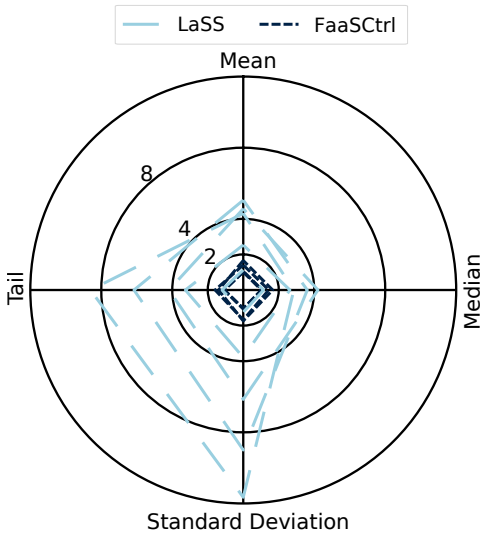
In our formulation, we have 5 hyperparameters: τ, a, b, P_{step} and C_{step}. The *fairness* constraint ensures that the S^{fair} metric exceeds the value τ. a and b are the weights of the S^{fair} and S^{perf} metrics in the reward function. P_{step} and C_{step} alter an application's real-time priority and the assigned number of cores by a fixed step size. We start out by setting the value of C_{step} to 1 (similar to [18]). We then use the one-at-a-time (OAT) hyperparameter tuning approach to find the best values for the remaining hyperparameters. One-at-a-time hyperparameter tuning involves adjusting one hyperparameter at a time while keeping others constant to observe its impact on the model's performance [97]. This method helps systematically identify the optimal value for each hyperparameter.

The order of setting these hyperparameters is as follows. We set P_{step} first by running 5 random pairs of LS and BE applications with default values of a, b and τ (1, 1 and 0.5) and different values of P_{step}. We choose the one that produces the best average CL across LS applications. Similar to P_{step}, we set the hyperparameter values of a and b. For τ, we evaluate the comprehensive latency of BE applications, while ensuring that the CL of LS applications lies within the circle of radius 2. In Figure 9, we show that the tuple $\langle P_{step} = 10, a = 1, b = 1, \tau = 0.5 \rangle$ is one of the best configurations, and thus we choose it.

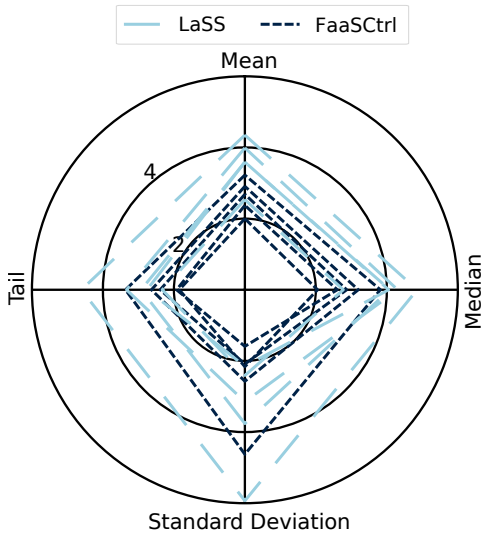
C. Evaluating Comprehensive Latency and Fairness

To evaluate the efficacy of our design, we measure the standard deviation, mean, tail, and median of the response latency (time between delivering a request and receiving the

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015

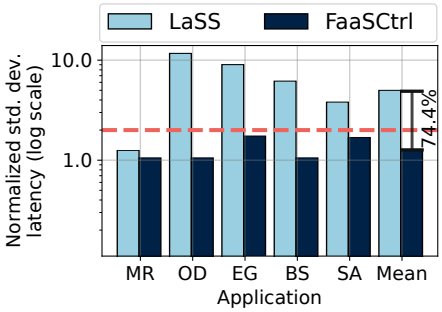


(a) CL of LS applications when colocated with VP.

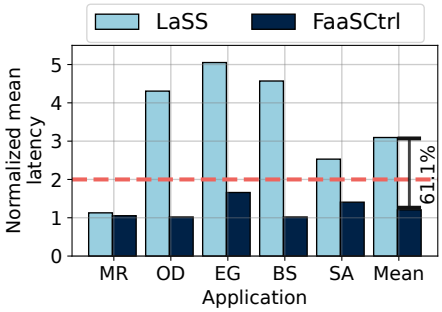


(b) CL of VP when colocated with LS applications.

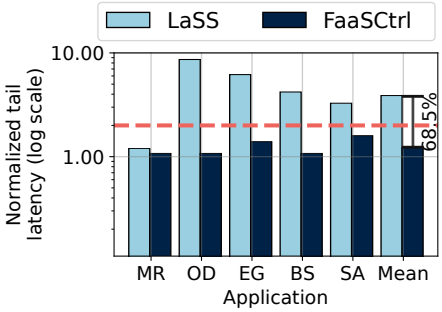
Fig. 10: The components of the comprehensive latency (CL) of an LS application when executed along with the VP application, which is an BE application. (normalized to *isolated* execution).



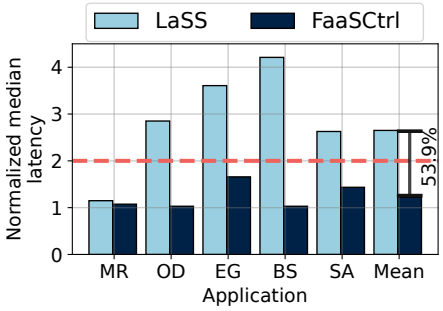
(a) The standard deviation in the response latency of LS applications.



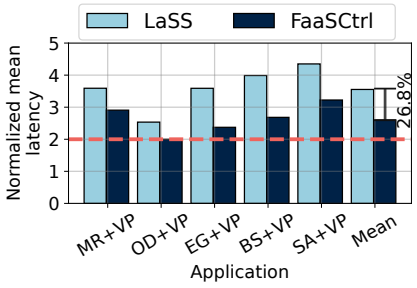
(b) The mean response latency of LS applications.



(c) The tail response latency of LS applications.



(d) The median response latency of LS applications.



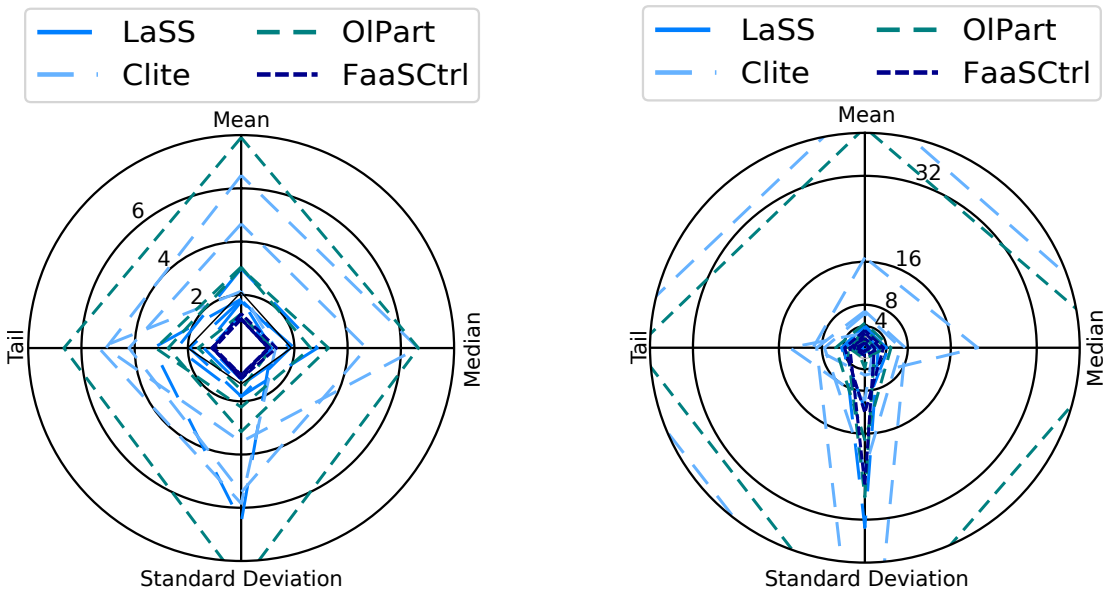
(e) The mean response latency of VP when colocated with an LS application.

Fig. 11: Statistics related to the response latency of applications when an LS application is executed along with the VP application, which is an BE application. (normalized to *isolated* execution).

response) of an LS application. Subsequently, we compare our solution against LaSS, which is specifically designed for serverless platforms. Recall that the degradation in the execution latency of LS applications is maximum when colocated with the VP application. For this study, VP was chosen as the BE application. Figure 10 shows the components of the comprehensive latency of *FaaSCtrl* and LaSS [7]. Later on, we compare the latency components of all the LS applications when colocated with all the BE applications (discussed in Section VIII-C1).

We observe that *FaaSCtrl* improves the standard deviation, mean, tail and median of the response latency of LS applications by 74.4%, 61.1%, 68.5% and 53.9% (resp.), on average (normalized to LaSS) (see Figure 11). Furthermore,

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015



(a) CL of LS applications when colocated with BE applications.

(b) CL of BE applications when colocated with all LS applications.

Fig. 12: The components of the comprehensive latency (CL) of applications when all LS applications are executed along with all BE applications (normalized to *isolated* execution).

we observe that an application's comprehensive latency is within the CL boundary (see Figure 10). This is happening because our RL algorithm is able to find the optimal resource configuration by exploring the search space efficiently. As a result, the CPU wait time, *#nvcs* events, and *#L1D_MPKI* reduce by up to 99.3%, 99.6%, and 6.4%, respectively. Effectively, our RL algorithm is eliminating all non-voluntary context switches because the time slice of SCHED_RR is 100 msec, whereas the range in SCHED_OTHER is between 0.75 msec and 6 msec [98]. Linux prioritizes SCHED_RR scheduling over SCHED_OTHER scheduling, which reduces the LS application's CPU wait time (total time spent in the run queues) to a near-zero value. Consequently, BE applications are pre-empted because of their lower priority.

The mean response latency of the VP application improves by 26.8%, on an average (as compared to LaSS) (as shown in Figure 11e). This is because the mean wait time (time spent waiting in Apache Openwhisk) of requests decreases by up to 79.8%.

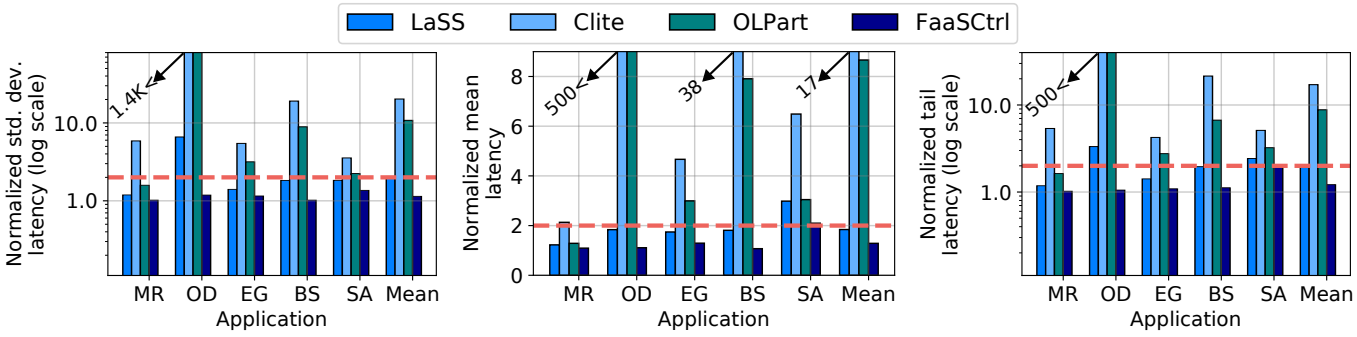
1) All 25 combinations: $LS \times BE$: Let us now consider all pairwise combinations of LS and BE workloads. The standard deviation, mean, tail, and median latency of the response time for LS applications reduce by 71.6%, 63.6%, 67.8%, and 58.1% (resp.), on an average, as compared to LaSS. Furthermore, *FaaSCtrl* is able to satisfy the CL boundary of all combinations. For the BE applications, we observe that *FaaSCtrl* reduces the mean response latency by 16%, on an average.

D. Evaluating Scalability

To evaluate the scalability of *FaaSCtrl*, we measure the standard deviation, mean, tail and median of the response latency while executing all five LS applications along with all five BE applications simultaneously. We run all the LS and BE applications with a moderate fraction of their maximum arrival rate (between 5% and 15%). Figure 12 shows a comparison between all the components of the comprehensive latency (CL) of LaSS [7], Clite [19], OLPart [21] and *FaaSCtrl*. We observe that *FaaSCtrl* performs better in all the components of CL compared to others and is within the CL boundary. Clite and OLPart perform poorly due to their inability to mitigate CPU contention. Among the LS applications, the object detector application suffers the highest degree of resource contention. This is because the resource partitioning schemes assign a fixed number of CPU cores (1-3) to the container processes of OD that share a file-level futex lock on *libcaffe2.so* of the py-torch library. As a result, the container processes and their threads sometimes wait for a long time to acquire the lock, thereby increasing the number of voluntary context switch events and CPU wait time by 1.35× and 4.3K ×, respectively. For the rest, the CPU wait time, and *#nvcs* events increase by up to 8.7× and 1.1× compared to isolated execution, respectively.

In comparison to LaSS, *FaaSCtrl* reduces the standard deviation, mean, tail and median of the response latency of LS applications by 44.6%, 30.2%, 36.9% and 27.7% (resp.), on an average (see Figure 13). We observe that the CPU wait time, *#nvcs* events, *#L1D_MPKI* and *#L2_MPKI* in LS applications reduces by up to 99.5%, 99.3%, 11.7% and 12.7% as compared to LaSS. We also observe that the mean response latency of BE applications improved by 17.9% on an average, as compared to LaSS (as shown in Figure 13e). We observe improvement in the response latency because the mean wait time of requests decreases by up to 45.8%.

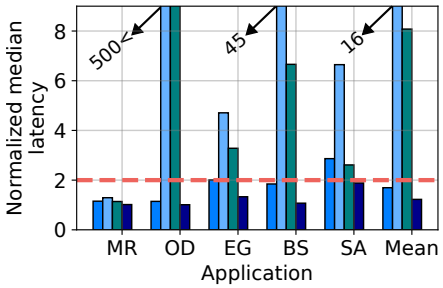
This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2024.3473015



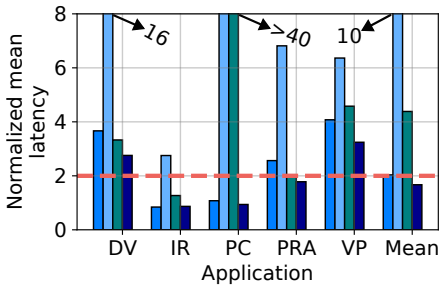
(a) The standard deviation in the response latency of LS applications.

(b) The mean response latency of LS applications.

(c) The tail response latency of LS applications.



(d) The median response latency of LS applications.



(e) The mean response latency of BE applications.

Fig. 13: Statistics related to the response latency of applications when all LS applications are running along with all BE applications (normalized to an *isolated* execution).

TABLE VII: The impact of different numbers of BE applications on LS applications in *FaaSCtrl* when compared against LaSS [7].

Configuration	BE application	Improvement in LS applications				Improvement in BE applications
		Mean	Median	Tail	Std. Dev.	Mean
5LS + 1BE	IR	28.4%	29.1%	31.8%	36.0%	13.8%
5LS + 2BE	IR, DV	20.4%	19.3%	29.4%	33.1%	11.3%
5LS + 3BE	IR, DV, PC	21.7%	17.8%	24.7%	36.6%	13.6%
5LS + 4BE	IR, DV, PC, PRA	23.1%	17.5%	27.5%	40.6%	2.8%
5LS + 5BE	IR, DV, PC, PRA, VP	30.2%	27.7%	36.9%	44.6%	17.9%
Mean		24.5%	21.7%	29.8%	37.9%	10.1%

E. Evaluating Impact of BE applications

To capture the influence of different numbers of BE applications on LS applications, we colocate 5 LS applications with different numbers of BE applications. In this experiment, we colocate 5 LS applications with varying numbers of BE applications, starting by adding the BE application with the shortest service time and incrementally including BE applications with progressively longer service times. This approach allows us to systematically observe the impact of increasing BE application service times on the performance of LS applications. We observe that *FaaSCtrl* improves the mean, median, tail, and standard deviation of the response latency of LS applications by 24.5%, 21.7%, 29.8%, and 37.9% when compared against LaSS [7] on average, respectively (see Table VII). This is because by controlling the real-time priority values and the assigned number of cores, we reduce the contention that LS applications experience. On the other hand, *FaaSCtrl* improves the mean response latency of BE applications by 10.1%. This is because the waiting time for requests has improved.

F. Limit Study

To determine the system load at which the comprehensive latency of an LS application remains within acceptable bounds, we conduct a limit study on *FaaSCtrl* to identify the breakdown point. In this analysis, we incrementally increase the number of LS applications running on the system while maintaining a constant number of BE applications (i.e., 5). We then evaluate the average comprehensive latency of LS applications to ensure that their components remain within the CL boundary, using a Kiviat plot for validation. Figure 14 illustrates that *FaaSCtrl* successfully maintains the comprehensive latency of LS applications within the CL boundary until 13 applications (8 LS + 5 BE) are running on the system. Additionally, *FaaSCtrl* ensures the overall system's fairness (figure not shown due to space constraints).

IX. Conclusion

In this paper, we introduce a novel intra-node resource manager that limits the comprehensive latency of an application within the CL boundary, while ensuring overall system's

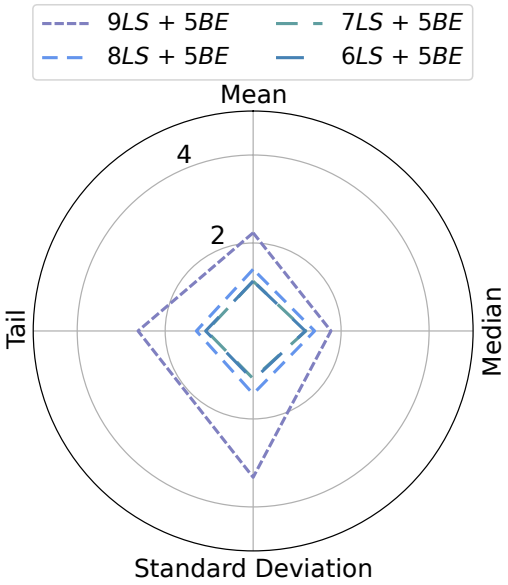


Fig. 14: The average comprehensive latency of LS applications when varying numbers of LS applications are colocated with 5 BE applications (normalized to their *isolated* execution).

fairness. To the best of our knowledge, this has not been done before in the area of serverless applications. To do so, we had to perform a detailed characterization study and identify the metrics of interest such as L1 cache based events, L2 cache based events, #nvcs events and the CPU wait time. We design an RL-based scheme to assign real-time priority and CPU cores to an LS application. The reward function captures an application's performance and system's fairness using the CPU wait time and the IPC metric. Additionally, it penalizes the model if it generates an invalid resource configuration that violates our objective. Finally, we were able to show that our design is scalable; for 10 applications (5 LS + 5 BE), we reduce the standard deviation, mean, tail and median of the response latency LS applications by 44.6%, 30.2%, 36.9% and 27.7% on an average, as compared to LaSS [7].

REFERENCES

[1] G. R. Russo, T. Mannucci, V. Cardellini, and F. L. Presti, "Serverledge: Decentralized function-as-a-service for the edge-cloud continuum," in *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2023, pp. 131–140.

[2] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2020.

[3] Mahmoudi, Nima and Khazaei, Hamzeh, "Performance modeling of metric-based serverless computing platforms," *IEEE Transactions on Cloud Computing*, 2022.

[4] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya, "Dependent function embedding for distributed serverless edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2346–2357, 2021.

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[6] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: state-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[7] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 239–251.

[8] "Azure functions – serverless functions in computing — microsoft azure," https://azure.microsoft.com/en-us/products/functions/#overview, (Accessed on 10/07/2022).

[9] "Serverless computing - aws lambda - amazon web services," https://aws.amazon.com/lambda/#:~:text=AWS%20Lambda%20is%20a%20serverless,pay%20for%20what%20you%20use., (Accessed on 10/07/2022).

[10] "Cloud functions — google cloud," https://cloud.google.com/functions, (Accessed on 10/07/2022).

[11] "Ibm cloud functions," https://cloud.ibm.com/functions/, (Accessed on 10/07/2022).

[12] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[13] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 467–481.

[14] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.

[15] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[16] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[17] M. Golec, R. Ozturac, Z. Pooranian, S. S. Gill, and R. Buyya, "Ifaasbus: A security-and privacy-based lightweight framework for serverless computing using iot and machine learning," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 5, pp. 3522–3529, 2021.

[18] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[19] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[20] P. Pang, Q. Chen, D. Zeng, and M. Guo, "Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 441–456, 2020.

[21] R. Chen, H. Shi, Y. Li, X. Liu, and G. Wang, "Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 347–364.

[22] Amazon, "Coca-cola freestyle launches touchless fountain experience in 100 days using aws lambda — case study — aws," 2024. [Online]. Available: https://aws.amazon.com/solutions/case-studies/coca-cola-freestyle/

[23] ——, "Neiman marcus case study — aws amplify — aws," 2024. [Online]. Available: https://aws.amazon.com/solutions/case-studies/neimanmarcus-case-study/

[24] ——, "Case studies - optimizing enterprise economics with serverless architectures," 2024. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/optimizing-enterprise-economics-with-serverless/case-studies.html

[25] ——, "Netflix: Aws lambda case study," 2024. [Online]. Available: https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/

[26] ——, "Aws re:invent 2020: Building the next generation of residential robots - youtube," 2024. [Online]. Available: https://www.youtube.com/watch?v=-1PDC6UOFtE

[27] E. Piccinin, "Serverless functions for microservices? probably yes, but stay flexible to change," 2021. [Online]. Available: https://www.infoq.com/articles/serverless-microservices-flexibility/

[28] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[29] V. W. Reporter, "The value of a millisecond: Finding the optimal speed of a trading infrastructure — tabb group," 2008. [Online]. Available: https://research.tabbgroup.com/report/v06-007-value-millisecond-finding-optimal-speed-trading-infrastructure

[30] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search presentation," 01 2009.

[31] A. Singla, B. Chandrasekaran, P. B. Godfrey, and B. Maggs, "The internet at the speed of light," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 1–7.

[32] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 625–638.

[33] D. Poccia, "New – provisioned concurrency for lambda functions — aws news blog," 2022. [Online]. Available: https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/

[34] Nzthiago, "Azure functions premium plan — microsoft learn," 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#available-instance-skus

[35] Y. Su, D. Feng, Y. Hua, and Z. Shi, "Understanding the latency distribution of cloud object storage systems," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 71–83, 2019.

[36] N. Tanković, T. G. Grbac, and M. Žagar, "Elaclo: A framework for optimizing software application topology in the cloud environment," *Expert Systems With Applications*, vol. 90, pp. 62–86, 2017.

[37] A. Samanta and R. Stutsman, "A case of multi-resource fairness for serverless workflows (work in progress paper)," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, 2023, pp. 45–50.

[38] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 168–181.

[39] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 311–327.

[40] C. Denninnart, J. Gentry, and M. A. Salehi, "Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 6–15.

[41] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "Drlpart: a deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 175–188.

[42] C. Liu, F. Tang, Y. Hu, K. Li, Z. Tang, and K. Li, "Distributed task migration optimization in mec by extending multi-agent deep reinforcement learning approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1603–1614, 2020.

[43] D. Cui, Z. Peng, J. Xiong, B. Xu, and W. Lin, "A reinforcement learning-based mixed job scheduler scheme for grid or iaas cloud," *IEEE Transactions on Cloud Computing*, vol. 8, no. 4, pp. 1030–1039, 2017.

[44] X. Chen, F. Zhu, Z. Chen, G. Min, X. Zheng, and C. Rong, "Resource allocation for cloud-based software services using prediction-enabled feedback control with reinforcement learning," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1117–1129, 2020.

[45] C. Qiu, H. Yao, C. Jiang, S. Guo, and F. Xu, "Cloud computing assisted blockchain-enabled internet of things," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 247–257, 2019.

[46] S. Zhang, C. Wang, and A. Y. Zomaya, "Robustness analysis and enhancement of deep reinforcement learning-based schedulers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 346–357, 2022.

[47] P. Zhao, J. Tao, L. Kangjie, G. Zhang, and F. Gao, "Deep reinforcement learning-based joint optimization of delay and privacy in multiple-user mec systems," *IEEE Transactions on Cloud Computing*, 2022.

[48] C. Zhang, M. Yu, W. Wang, and F. Yan, "Enabling cost-effective, slo-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1765–1779, 2020.

[49] P. Cong, J. Zhou, M. Chen, and T. Wei, "Personality-guided cloud pricing via reinforcement learning," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 925–943, 2020.

[50] Y. Tao, J. Qiu, and S. Lai, "A hybrid cloud and edge control strategy for demand responses using deep reinforcement learning and transfer learning," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 56–71, 2021.

[51] K. Shuai, Y. Miao, K. Hwang, and Z. Li, "Transfer reinforcement learning for adaptive task offloading over distributed edge clouds," *IEEE Transactions on Cloud Computing*, 2022.

[52] X. Chen, L. Yang, Z. Chen, G. Min, X. Zheng, and C. Rong, "Resource allocation with workload-time windows for cloud-based software services: a deep reinforcement learning approach," *IEEE Transactions on Cloud Computing*, 2022.

[53] C. Song, G. Han, and P. Zeng, "Cloud computing based demand response management using deep reinforcement learning," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 72–81, 2021.

[54] "Apache openwhisk is a serverless, open source cloud platform," https://openwhisk.apache.org/, (Accessed on 01/30/2022).

[55] "Cold start / warm start with aws lambda — octo talks !" https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/, (Accessed on 01/30/2022).

[56] "Squeezing the milliseconds: How to make serverless platforms blazing fast! — by markus thömmes — apache openwhisk — medium," https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0, (Accessed on 01/30/2022).

[57] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 1–13.

[58] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[59] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[60] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "Faasrank: Learning to schedule functions in serverless platforms," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2021, pp. 31–40.

[61] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: introducing an adaptive function placement algorithm," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 203–213.

[62] R. A. The, "Reinforcement learning," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Reinforcement_learning#Introduction

[63] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2022, pp. 1–14.

[64] J. Bhandari, D. Russo, and R. Singal, "A finite time analysis of temporal difference learning with linear function approximation," in *Conference on learning theory*. PMLR, 2018, pp. 1691–1692.

[65] Q. Cai, Z. Yang, J. D. Lee, and Z. Wang, "Neural temporal-difference learning converges to global optima," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[66] V. R. Konda and J. N. Tsitsiklis, "Onactor-critic algorithms," *SIAM journal on Control and Optimization*, vol. 42, no. 4, pp. 1143–1166, 2003.

[67] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," *Advances in neural information processing systems*, vol. 30, 2017.

[68] Y. Li, C. Zhao, X. Tang, W. Cai, X. Liu, G. Wang, and X. Gong, "Towards minimizing resource usage with qos guarantee in cloud gaming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 426–440, 2020.

[69] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, 2021, pp. 167–181.

[70] R. B. Roy, T. Patel, and D. Tiwari, "Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 292–305.

[71] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using

bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 46–49, 2021.
[72] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 1–6.
[73] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," 2022.
[74] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM symposium on cloud computing*, 2019, pp. 158–164.
[75] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 282–291.
[76] D. K. Kim and H.-G. Roh, "Scheduling containers rather than functions for function-as-a-service," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 465–474.
[77] M. Szalay, P. Matray, and L. Toka, "Real-time faas: Towards a latency bounded serverless cloud," *IEEE Transactions on Cloud Computing*, 2022.
[78] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, 2019, pp. 1063–1075.
[79] A. Galstyan, K. Czajkowski, and K. Lerman, "Resource allocation in the grid using reinforcement learning," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, vol. 1. IEEE Computer Society, 2004, pp. 1314–1315.
[80] J. Han and S. Lee, "Performance improvement of linux cpu scheduler using policy gradient reinforcement learning for android smartphones," *IEEE Access*, vol. 8, pp. 11 031–11 045, 2020.
[81] S. S. Mondal, N. Sheoran, and S. Mitra, "Scheduling of time-varying workloads using reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 9000–9008.
[82] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 167–179.
[83] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing*, 2009, pp. 137–146.
[84] "sched_rr_get_interval(2) - Linux manual page," 2022. [Online]. Available: {https://man7.org/linux/man-pages/man2/sched_rr_get_interval.2.html}
[85] P. De, V. Mann, and U. Mittaly, "Handling os jitter on multicore multithreaded systems," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.
[86] J.-P. Stauffert, F. Niebling, and M. E. Latoschik, "Reducing application-stage latencies for real-time interactive systems," in *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE, 2016, pp. 1–7.
[87] Y. Wei, "Research on real-time improvement technology of linux based on multi-core arm," in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. IEEE, 2021, pp. 1061–1066.
[88] X. Wang and J. F. Martínez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 113–125.
[89] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[90] "chrt(1) - linux manual page," 2022. [Online]. Available: https://man7.org/linux/man-pages/man1/chrt.1.html
[91] "taskset(1) - linux manual page," 2022. [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html
[92] C. Zhong, Z. Lu, M. C. Gursoy, and S. Velipasalar, "A deep actor-critic reinforcement learning framework for dynamic multichannel access," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1125–1139, 2019.
[93] D. Wang, H. Qin, B. Song, X. Du, and M. Guizani, "Resource allocation in information-centric wireless networking with d2d-enabled mec: A deep reinforcement learning approach," *IEEE Access*, vol. 7, pp. 114 935–114 944, 2019.
[94] Y. Sun, M. Peng, and S. Mao, "Deep reinforcement learning-based mode selection and resource management for green fog radio access networks," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1960–1971, 2018.
[95] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.
[96] T. Zahavy, M. Haroush, N. Merlis, D. J. Mankowitz, and S. Mannor, "Learn what not to learn: Action elimination with deep reinforcement learning," *Advances in neural information processing systems*, vol. 31, 2018.
[97] T. Trithipkaiwanpon and U. Taetragool, "Sensitivity analysis of random forest hyperparameters," in *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE, 2021, pp. 1163–1167.
[98] "fair.c - kernel/sched/fair.c - linux source code (v6.2.11) - bootlin," (Accessed on 03/30/2023).

Abhisek Panda received a bachelor's degree in computer science and engineering from Odisha University of Technology and Research, Bhubaneswar, Odisha.

He is currently a Research Scholar with the computer science and engineering deparment, Indian Institute of Technology Delhi, New Delhi, India. His current research interests include operating system, Intel SGX, and distributed computing.

Smruti R. Sarangi received the Ph.D in computer science from the University of Illinois at Urbana Champaign(UIUC), USA in 2006, and a B.Tech in computer science from IIT Kharagpur in 2002. After completing his Ph.D he has worked in Synopsys Research, and IBM Research Labs.

He has filed five US patents, seven Indian patents, and has published 120 papers in reputed international conferences and journals. He has published two popular books on computer architecture: (1) "Computer Organisation and Architecture", with McGrawHill in 2014 and later with WhiteFalcon in 2021 ("Basic Computer Architecture") , (2) and "Advanced Computer Architecture", with McGrawHill in 2021. He is currently the Associate Editor of the Elsevier Journal of Systems Architecture. He takes an active interest in teaching and technology-enhanced learning. He is currently the HoD of the Educational Technology Services Center. He has gotten the teaching excellence award in IIT Delhi in 2014 and is the latest recipient of the ACM Outstanding Contributions to Computing Education Award (OCCE, 2022). He got the Qualcomm Faculty Award in 2021 and has numerous best paper awards and nominations to his credit.