Utility-based Allocation of Industrial IoT Applications in Mobile Edge Clouds

Amardeep Mehta[*], Ewnetu Bayuh Lakew[*], Johan Tordsson[*], and Erik Elmroth[*]

Abstract

Mobile Edge Clouds (MECs) create new opportunities and challenges in terms of scheduling and running applications that have a wide range of latency requirements, such as intelligent transportation systems, process automation, and smart grids. We propose a two-tier scheduler for allocating runtime resources to Industrial Internet of Things (IIoTs) applications in MECs. The scheduler at the higher level runs periodically – monitors system state and the performance of applications – and decides whether to admit new applications and migrate existing applications. In contrast, the lower-level scheduler decides which application will get the runtime resource next. We use performance based metrics that tells the extent to which the runtimes are meeting the Service Level Objectives (SLOs) of the hosted applications. The *Application Happiness* metric is based on a single application's performance and SLOs. The *Runtime Happiness* metric is based on the Application Happiness of the applications the runtime is hosting. These metrics may be used for decision-making by the scheduler, rather than runtime utilization, for example.

We evaluate four scheduling policies for the high-level scheduler and five for the low-level scheduler. The objective for the schedulers is to minimize cost while meeting the SLO of each application. The policies are evaluated with respect to the number of runtimes, the impact on the performance of applications and utilization of the runtimes. The results of our evaluation show that the high-level policy based on *Runtime Happiness* combined with the low-level policy based on *Application Happiness* outperforms other policies for the schedulers, including the bin packing and random strategies. In particular, our combined policy requires up to 30% fewer

[*]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {amardeep, ewnetu, tordsson, elmroth}@cs.umu.se

runtimes than the simple bin packing strategy and increases the runtime utilization up to 40% for the Edge Data Center (DC) in the scenarios we evaluated.

Keywords: Mobile Edge Clouds, Edge Computing, IoTs, Distributed algorithms

1 Introduction

Mobile Edge clouds (MECs) [1], based on 5G technology, create new opportunities and challenges for latency-critical applications in domains such as intelligent transportation systems, process automation, and smart grids [2]. Some of the opportunities provided by MECs include improved end-user experience, reduced latency, and increased reliability and throughput. However, these opportunities bring associated challenges. It is not clear, for example, how one can cost-efficiently deploy and manage a large number of these applications given the heterogeneity of devices, application performance requirements, and variations in workload [1, 3].

Due to the limited capacity of MECs [4], it is impractical to reserve enough capacity at the edge to meet the resource demands of all Internet of Thing (IoT) applications. At any particular time the resource demands of IoT and Industrial Internet of Thing (IIoT) applications may exceed the available capacity at the edge, leading to infrastructure overload. This may in turn result in performance degradation of services and disgruntled end-users. In traditional cloud datacenters, several techniques such as elasticity, replication, and load balancing have been suggested to mitigate such issues, provided there is spare capacity in the infrastructure [5, 6]. However, such solutions cannot be directly applied in MEC due to the limited reserves of resources they have. Moreover, some applications such as mission-critical IoT applications cannot tolerate any disruption at all.

Therefore, it is particularly important for MECs to use efficient and effective schedulers, in order to maximize the use of available resources. To address this requirement, we propose a two-level scheduler in which the high-level scheduler periodically addresses long-term issues, such as Service Level Objective (SLO) violations and the low-level scheduler addresses short-term issues, such as intermittent resource shortages affecting particular IoT applications. The objective of the combined scheduler is to minimize operational monetary cost while meeting Quality of Service (QoS) requirements based on service differentiation principle. Thus we assume applications can be classified into different priority levels based on their respective Key Performance Indicators (KPIs). Specifically, IoT applications with stringent performance requirements are given higher priority than applications with comparatively lower performance requirements. Each IoT application is assigned to a Service Level Agreement (SLA) class (such as gold, silver and bronze), signifying its relative importance or criticality, and each class is assigned a penalty weight that

quantifies the cost of violating the performance requirements of an application in that class (See [7, 8]). This makes it possible to make decisions based on the cost of violating performance requirements and a scheduler can attempt to maintain the SLOs of higher-class applications by shifting resources from less performance-sensitive applications during periods of infrastructure overload or when allocating resources to new applications.

The high-level scheduler decides whether it can accept a new application on a runtime without violating performance requirements of all the applications hosted on that runtime and which application(s) to migrate or terminate to resolve long-term performance violations. The low-level scheduler selects which application will get the runtime resources next after execution of the current application terminates. It also ensures fairness among applications that belong to the same priority class.

Both schedulers use a feedback loop that takes into account both the performance of applications and resource utilization during their respective decision-making processes. For the high-level scheduler, each physical node reports an abstraction called the *runtime happiness value*, which indicates the satisfaction of applications running in the runtime with respect to their performance targets (This is inspired by [7, 8]). By sorting nodes according to their happiness value, the high-level scheduler can determine (i) which nodes are overloaded and thus need the number of applications reduced, either by migration or termination of applications, and (ii) which nodes are lightly loaded so that new applications can be accepted. The low-level scheduler uses an abstraction called *application happiness value* that indicates how actual performance deviates from expected performance. Using this abstraction, it decides which application gets the resource next.

The main contributions of our work are:

1. the design and evaluation of a hierarchical scheduler for resource allocation in Mobile Edge Clouds; and

2. the introduction of a new scheduling policy based on application happiness for the low-level scheduler and runtime happiness for the high-level scheduler.

2 Problem Definition

The performance requirements of IoT and IIoT applications vary greatly, as do their tolerance of SLO violations. Mission-critical IoT applications such as *autonomous car* cannot tolerate any delay in decision-making as it may lead to fatal accidents. On the other hand some applications, such as *video streaming*, are more tolerant of intermittent performance violations as long as average performance over time is acceptable. Due to the limited capacity at the edge and the variations in workload

Table 1: *Application Classes based on Performance Criticality*

Applications (category)	SLO (ms)	Weight	Tail (percentile)
Gold	100	4	99
Silver	100	2	95
Bronze	100	1	90

between IoT applications, the performance of critical applications may be affected unless an effective scheduling mechanism is in place.

We assume each IoT and IIoT application has a SLA that stipulates an SLO and a class – such as gold, silver and bronze – indicating how tolerant the application is to slight performance violations. Table 1 illustrates how applications could be classified into different priority groups according to their latency requirements [9, 10, 11]. For example, *autonomous car*, with its strict latency requirements, might be placed in the Gold category, whereas video streaming applications could be placed in the Bronze category. The intuition behind having applications run with different priority requirements is that resources can be taken from less critical applications to satisfy the demands of a more critical application. Similarly, less critical applications can be migrated to remote sites when the resource demand at the edge is at its peak. An application SLO can be expressed by associating desired performance – such as tail latency – with a percentage, indicating the proportion of requests that can meet the desired performance in a time window.

This work considers the problem of allocating IoT and IIoT applications to runtimes in a MEC ecosystem in order to minimize cost while meeting their QoS requirements. The goal is to continuously adjust the scheduling of applications, so as to ensure the applications meet their respective SLO targets as far as is possible given capacity constraints. When sufficient resource capacity is available in the infrastructure, each application is allocated sufficient resources to meet its performance target. The two-level scheduler allocates applications to runtimes, allowing several applications to execute on a single runtime, while minimizing cost and meeting the applications' performance requirements. The low-level scheduler resolves temporary capacity shortages by reallocating resources from low priority application to high priority applications. The high-level scheduler resolves long term capacity shortages by migrating offending applications either to a remote site or to other less loaded runtimes at the edge. Figure 1 shows the typical infrastructure and application deployment scenario we consider in this work.

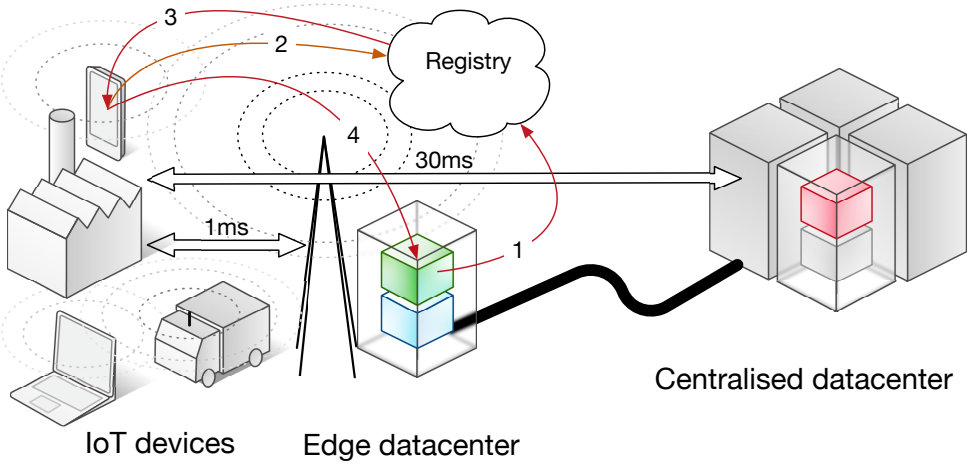

Figure 1: Mobile Edge Cloud infrastructure

3 System Model and Notation

In this section, we describe a general framework for IoT applications. The framework abstracts away the heterogeneity of hardware resources and provides a single environment in order to easily deploy and manage IoT applications.

3.1 Application model

Our model for IoT applications is based on Actor and Flow-Based Programming models [12][13][14]. An application can consist of one or more than one actors. A single functional unit, such as processing unit, is called an *actor*. An actor in an application do not share state with other actors and may represent sensor devices, computation units, or service units. Actors communicate with one another via *tokens* passed over their ports. An application can be written by simply connecting the ports of relevant actors.

Actor model: Each actor may have multiple in-ports and out-ports where the tokens are dequeued in First In First Out (FIFO) order for processing. For simplicity, we consider one in-port and one out-port per actor. An actor's properties are profiled over an observation period to help the schedulers in the placement decision making.

Let the set of actors in the system be $A = \{a_n \mid n = 1, 2, \ldots, N\}$. Each actor, $a_n, n \in 1, 2, \ldots, N$, has the following properties:

- **position** $p_n \in \{1, \ldots, I\}$, a number indicating the runtime to which an actor has been allocated;
- **service time** s_n, a number specifying the average computation time required by the actor to execute an action over the observation period;
- **token size** c_n, a number specifying the average size of the token (in bytes) over the observation period;
- **arrival rate** λ_n, a number specifying the average number of tokens arriving at the actor a_n per time unit over the observation period; and
- **priority** w_n, a number specifying the weight for an application.

Dataflow model: When connected in sequence via ports, actors create a flow of tokens from a source to a destination. The port connections between actors reflect the dataflow within the application. An application can consist of one or more dataflows. A dataflow may consist of one or more fixed actors (e.g. temperature sensor) and *floating* actors (e.g. data processing actor). Floating actors can migrate to other runtimes whereas fixed actors can not. In this work, we consider one floating actor and one fixed actor per dataflow of an application. A dataflow has the following properties:

- **delay requirement** $\overline{d_n}$, a number specifying maximum allowed round trip time between the fixed and the floating actor. This requirement defines SLO for the application.

For simplicity, we use one dataflow per application for our experiments.

3.2 Infrastructure model

The infrastructure is modelled as a graph $G = (V, E)$ where each vertex represents a computational resource and each edge represents a network link.

$$\begin{aligned} V &= \{v_i \mid i = 1, 2, ..., I\}, \\ E &= \{e_j \mid j = 1, 2, ..., J\}, \end{aligned} \tag{1}$$

Runtime model: Runtime resources are shared among the applications for executing their requests. A runtime can be modelled as a M/G/1 queue. A scheduler on the runtime can have different policies for allocating resources to the applications, such as Round Robin (RR), First Come First Serve (FCFS), Non-preemptive (NP) or Earliest Deadline First (EDF). We propose a new dynamic policy, Application Happiness (AH), which is based on how satisfied the application is with the runtime. We describe this policy in Section 5.1.

A Runtime v_i, $i \in \{1, 2, \ldots, I\}$, in the graph has two key properties:

- **location** q_i, a number specifying whether it is running on a device, edge, or a centralized DC; and
- **processing speed** α_i, a number specifying the relative processing speed of the runtime with respect to a standard processor's speed.

In addition to the above properties, a runtime v_i, $i \in \{1, 2, \ldots, I\}$, is associated with the following operational cost:

- **runtime usage cost** ζ_{q_i}, a function of the used processor time that returns the runtime's running cost.

We denote the costs for edge and central DCs as ζ_{edge} and $\zeta_{central}$, respectively. The cost of running an actor on a runtime at an edge DC will be higher than that for running it in a central DC.

Network model: Each link e_j, $j \in \{1, 2, ..., J\}$, in the graph has the following properties:

- **queuing latency** β_j, a number specifying the average round trip queuing delay due to routers or switches over a network link j; and
- **transfer rate** $\overline{b_j}$, a number specifying the bandwidth over link j.

To quantify β_j, we measured the Round Trip Time (RTT) for lambda AWS endpoints [15] from Umeå University, as shown in Figure 2. These measurements showed that the lowest achievable queuing latency was 20-40 ms for the round trip from Umeå to the EU-central-1 location (Frankfurt).

In addition, each edge has a **link usage cost** η_j, which is a function of bandwidth usage that returns the link's running cost.

4 System Dynamics

In this section, we study the system parameters that affect the allocation of applications to runtimes by the scheduler.

4.1 Cost dynamics

According to our previous studies [1, 4], the operational cost of a CPU-second decreases non-linearly with the number of servers in the DC as shown in Figure 3. For example, edge DCs with 15 and 150 servers would have compute costs of \$0.00004 and \$0.00003, respectively for a CPU-sec, whereas a central DC with 15000 servers would have a cost of \$0.00002. These costs are consistent with the prices offered by major cloud providers. For example, the costs of renting a 1 GB machine for one

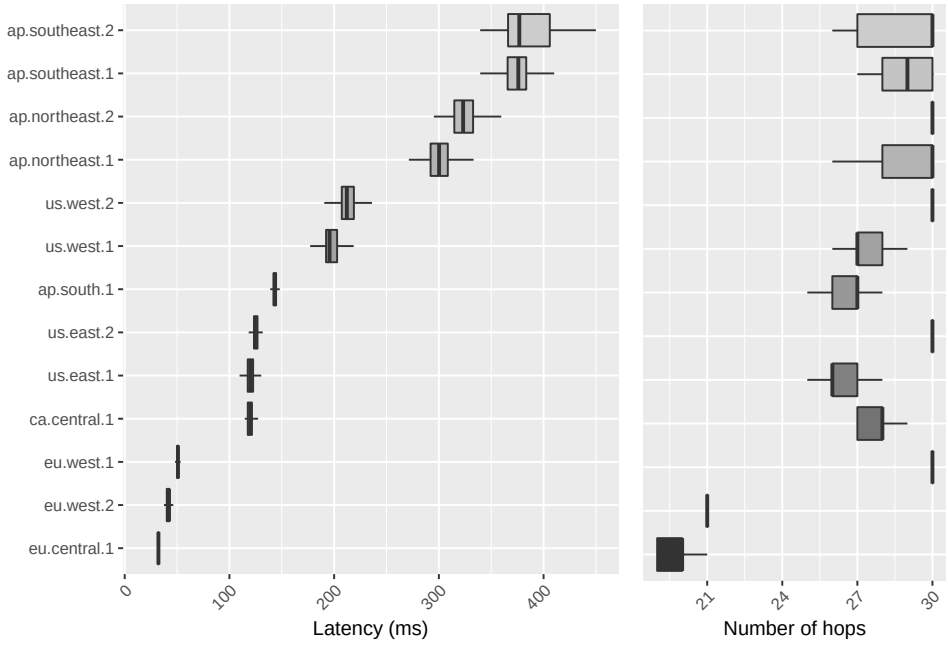

Figure 2: Round Trip Time (RTT) and number of hops for selected lambda AWS endpoints [15] from Umeå University.

Figure 3: Cost for a CPU-second in a data center

second for Amazon lambda and lambda@edge are $0.00001667 and $0.0000501, respectively [16].

Bandwidth pricing models for the transfer of data across the internal network of an Internet Service Provider (ISP) are generally proprietary and not publicly available. Some studies [17, 18, 19] suggest current bandwidth cost is around $0.03 per GB. However, these prices are falling over time.

4.2 Performance dynamics

Overall delays for an application's data (or token) are incurred at runtimes and at network links between the source and destination. Processing delays at runtimes can be challenging because of the *processor heterogeneity* of devices and *processor sharing* in the runtime. The M/G/1 queuing models can be used to understand the system time distribution for actors on a runtime. The system response time distribution depends on the service time variability of actors on the runtime and can be given using Pollaczek-Khintchine formula for M/G/1 queues [20]. Thus, a policy that reduces the variability of actors within a runtime can reduce the response time of actors hosted on that runtime. In our previous work [1], we proposed a *size-based policy* where actors in a service time range are allocated to runtimes in that service range. This policy was found to perform better than simple bin packing and random strategies.

5 The Two-level Scheduler and Happiness Metrics

In this section, we describe the two-level scheduler we have developed for allocating applications to runtimes in MECs. The schematic design of the scheduler is shown in Figure 4. Each application's performance is monitored periodically and reported to the scheduler in order to determine how its performance deviates from the relevant targets. We monitor application performance using a simple performance deviation model suggested by Lakew et al. [7].

5.1 Low-level scheduler

The low-level scheduler decides which application's request will obtain the runtime resource after completion of the current request. Existing policies, such as Round Robin (RR), Earliest Deadline First (EDF), First Come First Serve (FCFS) and Non-preemptive (NP), do not require feedback to allocate resources to the requests in the runtime's waiting queue.

We propose a feedback policy – the Application Happiness (AH) policy – where "happiness" is based on waiting time of requests in the runtime, priority, network latency and performance requirement(s). We write $h_{a_n}^{v_i}$ to denote the happiness

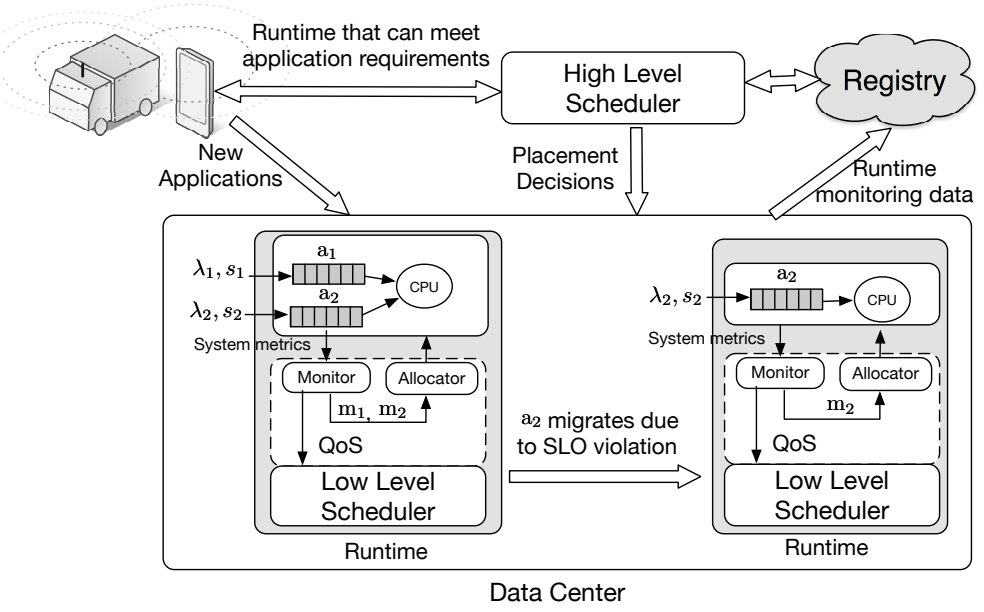

Figure 4: Two-level scheduler for application placement in Mobile Edge Clouds

of application a_n at runtime v_i. We compute the extra resources allocated to an application a_n after execution of a token on runtime v_i at time t using the following formula:

$$x^{v_i}_{a_n,t} = 1 - \frac{(w_n m_n + \beta_j + s_n)}{\overline{d_n}}, \tag{2}$$

where m_n is the measured waiting time for the executed token of application a_n on runtime v_i at time t.

Then $h^{v_i}_{a_n}$ is computed using the extra allocated resources at time t, $x^{v_i}_{a_n,t}$, and the happiness of the application at time $(t-1)$:

$$h^{v_i}_{a_n,t} = \theta x^{v_i}_{a_n,t} + (1-\theta) h^{v_i}_{a_n,t-1}, \tag{3}$$

We use exponential smoothing to adjust the relative importance of the two components, based on the parameter θ, where where $0 \leq \theta \leq 1$. Values of $\theta > 0.5$ give greater importance to $x^{v_i}_{a_n,t}$ than $h^{v_i}_{a_n,t-1}$; and the greater the value of θ, the greater the importance ascribed to $x^{v_i}_{a_n,t}$.

If $h^{v_i}_{a_n} < 0$ then the runtime is not meeting the application's targets, whereas $h^{v_i}_{a_n} > 0$ indicates that the runtime has allocated more resources than needed to meet the application's performance targets.

If the mean (or median) happiness value of an application becomes negative over a time window, then the runtime informs the high-level scheduler, which will then reschedule applications in order to mitigate the QoS violations. The overall goal of our scheduler is to allocate runtime resources in such a way that all applications have non-negative happiness values.

The happiness value of an application on a runtime is updated after the execution of a request (or token) associated with the application. The low-level scheduler then assigns the runtime resource to the application having the lowest happiness value among all the hosted applications.

5.2 High-level scheduler

The high-level scheduler is the cornerstone of our scheduling framework. It decides where a new application should be placed and which applications to migrate or terminate, by taking into account the current and predicted state of the system, the priority classes of applications and their respective SLOs against overall system cost. In case of a new request for resource allocation of an application, it places the application at a runtime of either edge DC or Centralized Data Center (CDC) depending on where application's operational cost is lower and the performance target of the application can be met [1]. It determines the optimal scheduling arrangement of applications on runtimes of a DC based on the happiness of applications.

In order to decide which application to migrate or where to place an application, the high-level scheduler determines the state of the edge DC and CDC using periodic feedback about the performance of applications and their respective priorities and targets. For example, the scheduler may decide to migrate an application whose targets are not being met or to migrate one or more low-priority applications from the edge DC to either CDC or back to the IoT devices.

The focus here is how to use an application's priority and SLO when making decisions in such a way that the performance of applications that have already been deployed is not compromised. Since applications behave differently under different configurations and applications with different performance requirements may enter or exit the DC at any time, there is a need to continuously adapt the system configuration based on the performance of hosted applications, hence our decision to use happiness metrics.

In order to decide where to place an application or which application to migrate, the high level scheduler uses *runtime happiness values*. The runtime happiness value h_{v_i} of runtime v_i is a function (typically mean or median) of the application happiness values $h_{a_n}^{v_i}$ for applications allocated to runtime v_i. The computation of runtime happiness from application happiness values is not the focus of this work. However, we did find that using the median instead of the mean, gives slightly better results.

Algorithm 1 Pseudo-code for the high-level scheduler

Require: $h_{a_n}^{v_i}$, Happiness value for each application a_n deployed at runtime v_i

```
1:  COMPUTE h_{v_i}, Happiness value for each runtime at the edge DC and CDC using h_{a_n}^{v_i}
2:  while h_{v_i} < 0 do
3:      while there are happy runtimes do
4:          SELECT application to MIGRATE based on priority and h_{a_n}^{v_i}
5:          MIGRATE to the most happy runtime on the same DC
6:      end while
7:      if there are still unhappy runtimes then
8:          SELECT application to MIGRATE based on priority and h_{a_n}^{v_i}
9:          MIGRATE to the most happy runtime of another DC or IoT devices
10:     end if
11: end while
12: if A new placement request is received then
13:     while there are happy runtimes do
14:         PLACE the new request at the most happy runtime of either edge DC or CDC where operational
            cost is lower while meeting the performance target
15:     end while
16:     if the DC cannot satisfy the request then
17:         MIGRATE back to the IoT device
18:     end if
19: end if
```

Once the happiness values for each runtime have been computed, the values can be sorted. Then the runtime with the highest happiness value becomes a potential candidate for the placement of new applications entering the system. The scheduler subsequently readjusts the configuration, based on runtime and application happiness values, to ensure that the SLOs of all applications are satisfied.

Algorithm 1 presents pseudo-code for the high-level scheduler. Applications are migrated from runtimes running unhappy applications to the happiest runtime (lines 3–6). Applications that should be migrated, are selected based on their runtime behavior and their priorities. If migration of applications within the same DC cannot resolve the issue, indicating the current DC can not fulfill the applications' performance targets, some applications are either migrated to another DC, or moved back to the IoT device (lines 7–10). A new application is allocated to a runtime with the highest happiness value (lines 12–18). Reshuffling of applications with minimum migration cost may be required when accepting new applications so that the infrastructure has near optimal system configuration.

6 Evaluation

In this section we describe the results of simulations we performed in order to evaluate our approach. We describe the simulation set-up in Section 6.1. The results of an extensive comparison of our proposed algorithm against state-of-the-art algorithms are presented in Section 6.2.

6.1 Simulation setup

We simulated both infrastructure and applications in order to study system performance as well as the performance of the different scheduling algorithms.

6.1.1 Infrastructure set-up

For our study, we simulated one edge DC and one CDC, as shown in Figure 1. The network RTT represents the queuing delay due to routers and switches on the packet transmission path. Based on the RTT measurement for the closest AWS lambda endpoint, as shown in the Figure 2, we set the simulated average and standard deviation of latency between devices and the central DC to 30 ms and 10 ms respectively. The simulated average and standard deviation of RTT latency between devices and the edge DC, based on the 5G specification [21][22][23], was set to 1 ms and 0.1 ms respectively. However, it should be noted that a recent study shows that RTT latency of around 10 ms is more realistic for edge DCs [24]. We ignored latency due to software stacks when simulating runtimes.

6.1.2 Applications

To capture arrival rate as well as data size variations, we consider several different application configurations, as summarized in Table 2. The service time distribution was modeled using a bounded Pareto distribution, which is very common in computing workloads [20]. The bounded pareto distribution has density function,

$$f(x) = \gamma x^{-\gamma-1} \frac{k^\gamma}{1 - (\frac{k}{p})^\gamma}, \tag{4}$$

where $k \leq x < p$ and $0 < \gamma < 2$. Based on the SLO (based on RTT in this case) listed in Table 1, the applications' mean service time requirements could vary between 1 ms and 100 ms and the standard deviation of service time is simulated as 10% of the mean value.

Table 2: *Application configurations based on arrival rate and data size [2].*

Config.	Application	Inter Arrival Time ($1/\lambda$) (ms)	Data size (Kb)
1	A0	100	0.5
2	A1	100	1000
3	A2	1000	1000

6.2 Experiments

In order to evaluate the performance of the schedulers and policies described in Section 5, we developed an event-based simulator in Python. We simulated 450 applications from nine configurations including three categories and three configurations as shown in Tables 1 and 2. The simulated applications were randomized before they entered the system. Each experiment was run 10 times and descriptive statistics, such as mean and standard deviation, of the scheduling algorithms' performance metrics were calculated.

The following list describes the different policies we evaluated for the high-level scheduler. All these policies also enforces the application's SLO constraints.

1. Bin Packing with Performance requirement at Least Utilized Runtime (BP_Perf_LUR): the application is allocated to the runtime that has the lowest utilization.
2. Bin Packing with Performance requirement with Runtime Happiness (BP_Perf_RH): the application is allocated to the happiest runtime.
3. Size Interval Actor Assignment in Groups at Least Utilized Runtime (SIAA_G_LUR): a group of runtimes serves applications that fall within a designated service range. The framework stores information about all the different groups in the registry based on their non-overlapping service ranges. The application is allocated to a runtime in the group that has the lowest utilization.
4. Size Interval Actor Assignment in Groups with Runtime Happiness (SIAA_G_RH): Similar to SIAA_G_LUR, except that the application is allocated to the happiest runtime.
5. Random with Performance requirement (Random_Perf): the application is allocated to a random runtime.

All these policies allocate a new runtime to the application if the current runtime fails to fulfill the performance requirements of the application over a given time window. We evaluated the high-level policies in combination with the low-level policies RR, NP, EDF, FCFS and AH (see the runtime model in Section III) for application placement.

We recorded the following metrics:

1. average number of runtimes required to place all applications;
2. mean and standard deviation of slowdown for all placed applications; and
3. mean and standard deviation of runtime utilization.

where the mean slowdown of an application was computed as the mean system waiting time divided by its service time.

For the experiments, we used a 10 second time window for event-based simulation. To evaluate the size-based polices, we defined 10 linear-spaced groups within service time range 1-100 ms [1]. We report our results for $\theta = 0.9$, as this was found to give slightly better results compared to other values for the exponential smoothing parameter.

6.2.1 Effect of service time distribution

In this case, we perform simulations to understand how the variation in service time distribution affects the number of runtimes required and the performance of applications. Lower values of γ, the bounded pareto distribution parameter (see Equation 4), correspond to higher variability in the service time distribution. Figure 5 shows the average number of runtimes required for the four high-level policies and five low-level policies. The top subplots show the average number of runtimes required for the Edge DC; the bottom subplots show the average number of runtimes required for the CDC.

Each subplot shows that the average number of runtimes required to place all the applications decreases as the value for the parameter γ increases. This happens due to the decrease in the variability of service times for applications.

Of the high-level policies based on runtime utilization, SIAA_G_LUR performed better compared to BP_Perf_LUR and Random_Perf. However, a high-level policy based on *runtime happiness*, such as SIAA_G_RH along with AH, outperformed other policies based on runtime utilization in terms of minimum number of runtimes required to place the applications, as shown in the Figure 5. Choosing a high-level policy based on runtime utilization, such as SIAA_G_LUR, resulted in up to a 20% reduction in the average number of runtimes required for placing applications. However, choosing a high-level policy based on runtime happiness, such as the combined policy SIAA_G_RH and AH, reduced the average number of runtimes by up to 30% at the Edge DC. There was a marginal decrease in the average number of runtimes required for the CDC as the service time variability in the workload decreases when the SIAA_G_RH and AH policies are combined. In all cases, the standard deviation for the number of runtimes was below 10% of the average number of runtimes required for placing all applications.

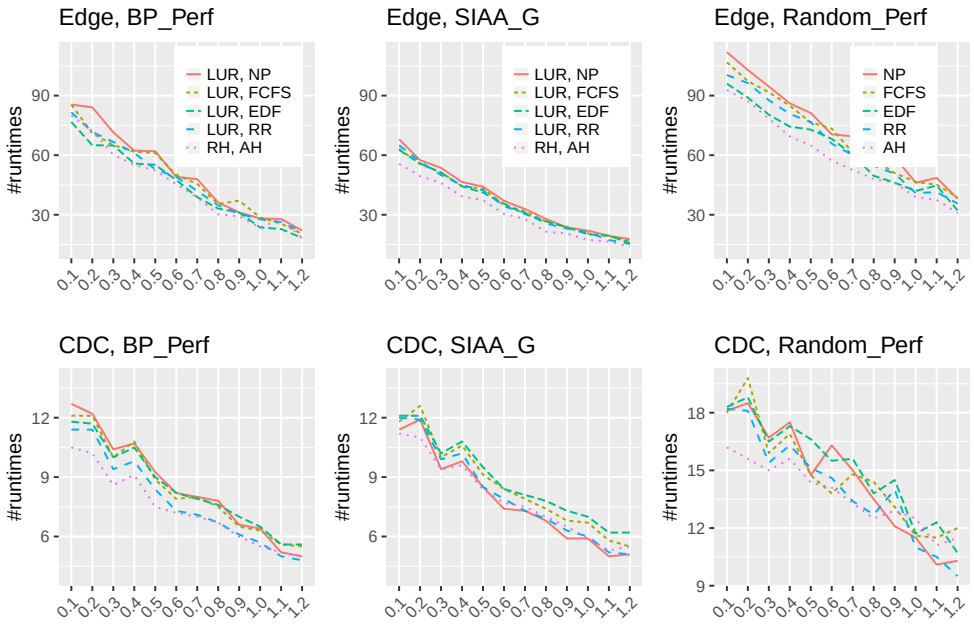

Figure 5: Mean number of runtimes as service time varies

The results for average application slowdown are shown in Figures 6. It can be seen that the slowdown increased sharply for the combined policy SIAA_G_RH and AH, compared to the other policies, as the application service time variability decreases. For example, the slowdown for this policy increased by 1.8 times as γ increased from 0.1 to 1.2. The slowdown, when $\gamma = 0.1$, for the combined policy SIAA_G_RH and AH was 1.5 and 1.6 times higher than SIAA_G_LUR and FCFS policy at the Edge and CDC, respectively; and 2.4 and 1.7 times higher when $\gamma = 1.2$.

The standard deviation of the slowdown also increased as the variability in the workload decreased, as shown in Figure 7. This is due to the fact that the proposed AH low-level policy allocates runtime resources among the applications so that the resources are not wasted by applications belonging to lower priority classes. The AH policy distributes the resources among applications better than other lower level policies.

The results for mean and standard deviation of runtime utilization are shown in Figures 8 and 9. For the Edge DC using the combined policy SIAA_G_RH and AH, the average and standard deviation of the runtime utilization was always higher than the others, whereas it is at similar level as the others for the CDC. For the Edge DC using the combined SIAA_G_RH and AH policy, the average runtime utilization

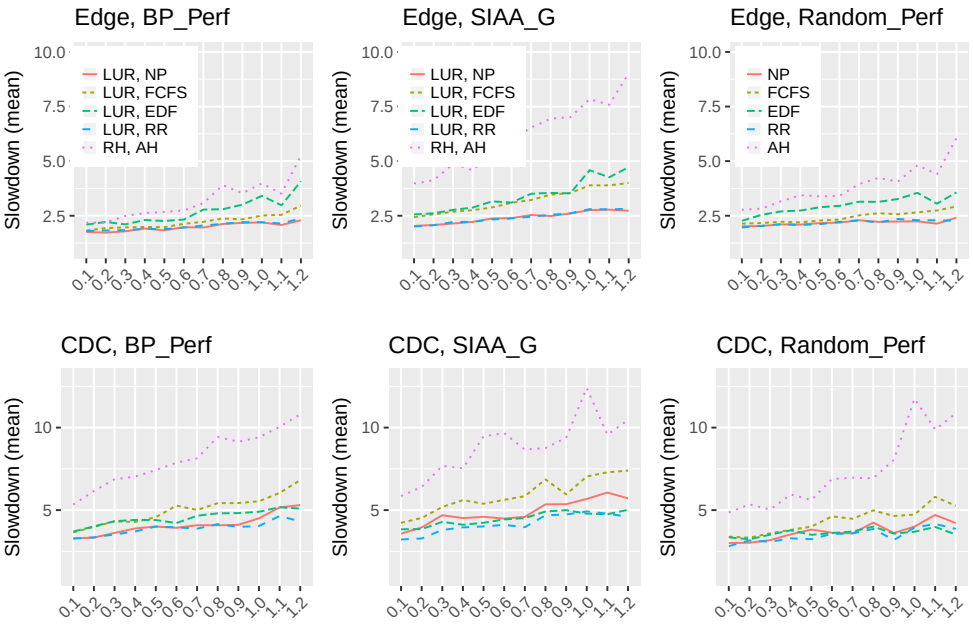

Figure 6: Mean slowdown of applications as service time varies

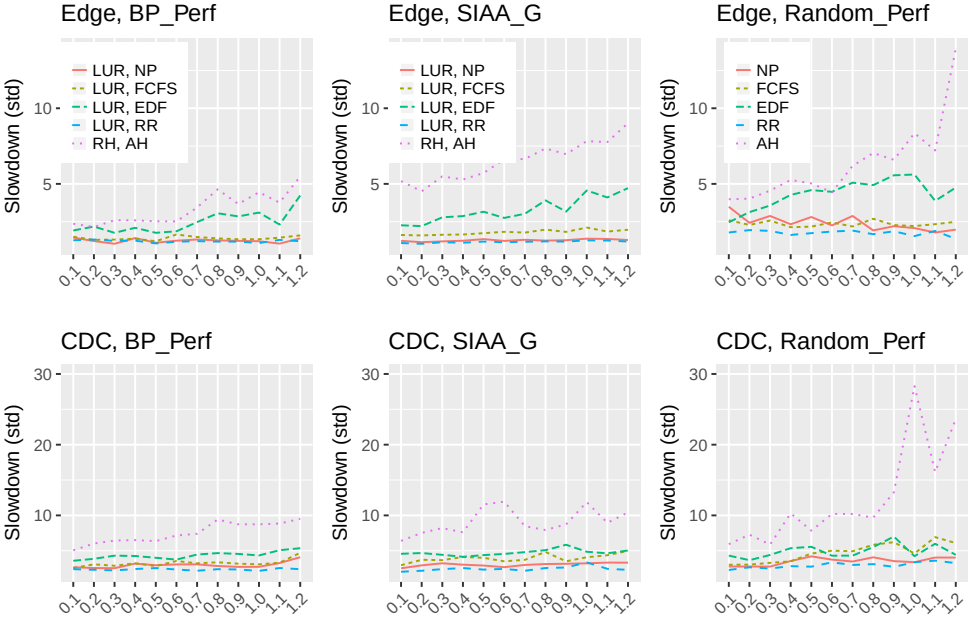

Figure 7: Standard deviation of slowdown as service time varies

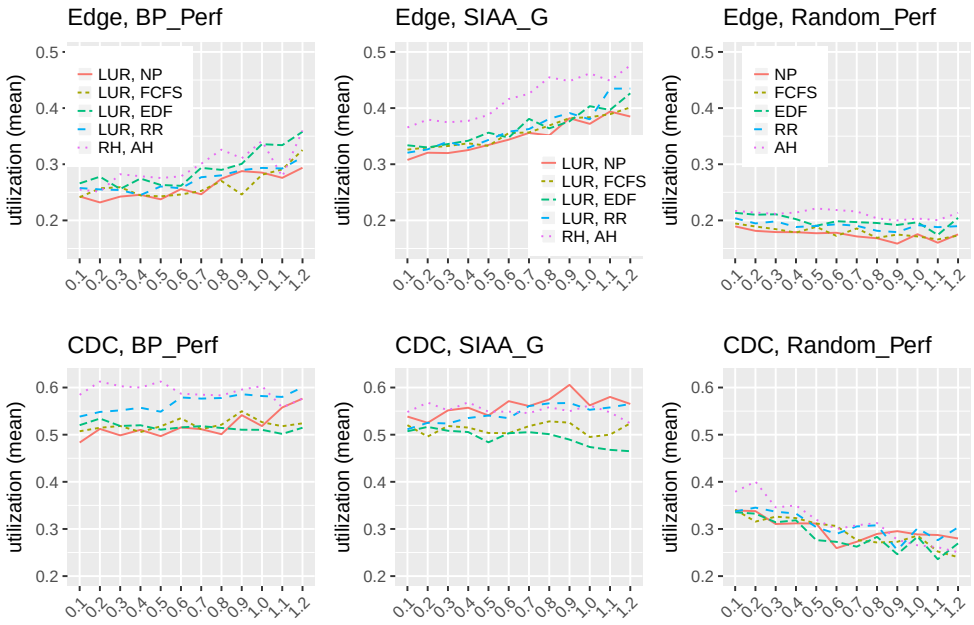

Figure 8: Mean utilization of runtimes as service time varies

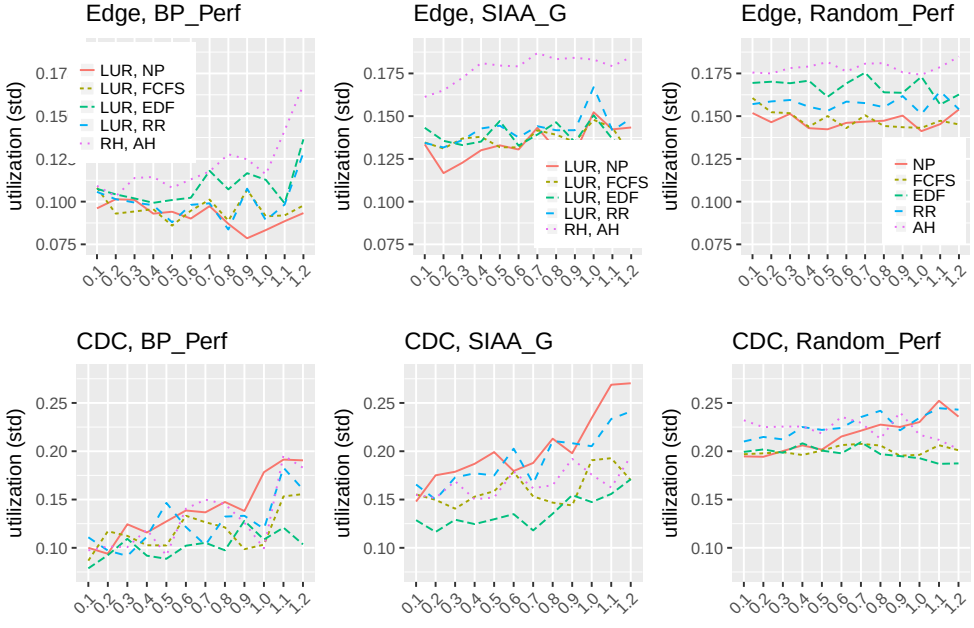

Figure 9: Standard deviation of utilization of runtimes as service time varies

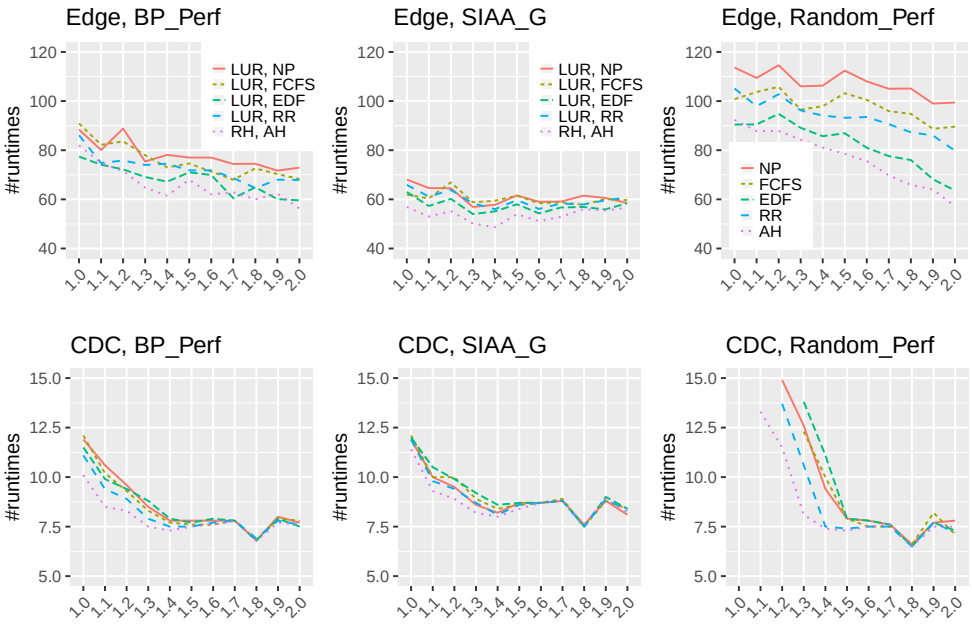

Figure 10: Mean number of runtimes as node speed increases ($\gamma = 0.1$)

increased from 35% to 50% as the service time variability parameter γ increased from 0.1 to 1.2. The average runtime utilization at the Edge DC using the combined policy SIAA_G_RH and AH was 5% higher than that of the SIAA_G_LUR and FCFS combined policy when $\gamma = 0.1$, whereas it was 10% higher when $\gamma = 1.2$.

Size-based policies, such as SIAA_G_LUR and SIAA_G_RH, increased the variation in runtime utilization compared to others, as shown in Figure 9. This is due to the fact that the runtimes are grouped based on the service time range when this policy is used, and applications having small service times are not stuck waiting due to applications having large service time.

For the CDC case, both the average and the standard deviation in the runtime utilization for the combined policy SIAA_G_RH and AH did not vary much compared to the others as service time varied.

6.2.2 Effect of varying the speed of MEC nodes relative to that of the IoT devices

In this experiment, the processing speed of the MEC nodes was increased by up to two times that of the IoT devices. Figure 10 shows the number of servers required at

the Edge DC and CDC when the server speed is increased in this way. As the node speed increased, the number of runtimes required to place the applications decreased for both Edge and CDC. The increase in server capabilities at the MEC, means it is beneficial to run more applications at the Edge because the applications' service times decrease (due to increased processing speeds at MEC). After a 40% increase in the server speed, the number of runtimes required to place the applications became constant. The SIAA_G_RH and AH combined policy required 15% fewer runtimes when the MEC node speed was 40% higher than that of the IoT devices for Edge DC.

Figures 11 and 12 show how the average and standard deviation of application slowdown, respectively, varied as the MEC node speed was increased. Both the average and the standard deviation of the application slowdown decreased as node speed increased. The average slowdown for the combination of the high-level policy SIAA_G_RH and the lower level policy AH remained constant for the Edge DC when the node speed is increased by up to 40%. After that, the slowdown decreased up to the same level of the average slowdown level of other policies. However the standard deviation sharply decreased to the same level of standard deviation of other policies when the node speed was increased by up to 40%.

Figure 13 shows the average runtime utilization at the Edge and CDC varied as the node speed increased. As the node speed increased the average utilization increased by up to 70% for all policies. The standard deviation for the combination of SIAA_G_RH and AH increased by 25% when the node speed increased by 40% as shown in the Figure 14.

6.2.3 Effect of Edge DC capacity

In this experiment, we investigated the effect of Edge DC capacity on the performance of the scheduling algorithms. Figure 3 shows that the compute cost decreased as the capacity of the Edge DC increased. The increase in capacity means that some applications, previously placed at the CDC, can now be allocated a runtime at the Edge, as the computation cost becomes cheaper at the Edge while bandwidth cost between Edge and CDC remains the same.

The average number of runtimes required was lowest for the (high-level) SIAA_G_RH policy. For the SIAA_G_RH high-level policy and AH low-level policy, there was around a 5% increase in the average number of runtimes required at Edge DC, whereas there is around a 5% decrease in the average number of runtimes for the CDC if the edge capacity is increased from 15 to 1500 servers. Increasing the edge capacity above 1500 does not increase or decrease the number of runtimes at the Edge or the CDC. The standard deviation for the number of runtimes required for placing all the applications is below 10% of the average number of runtimes for all the policies.

Both the average and standard deviation of application slowdown decreased for the Edge DC as the edge capacity increased, whereas it increases for the CDC. The

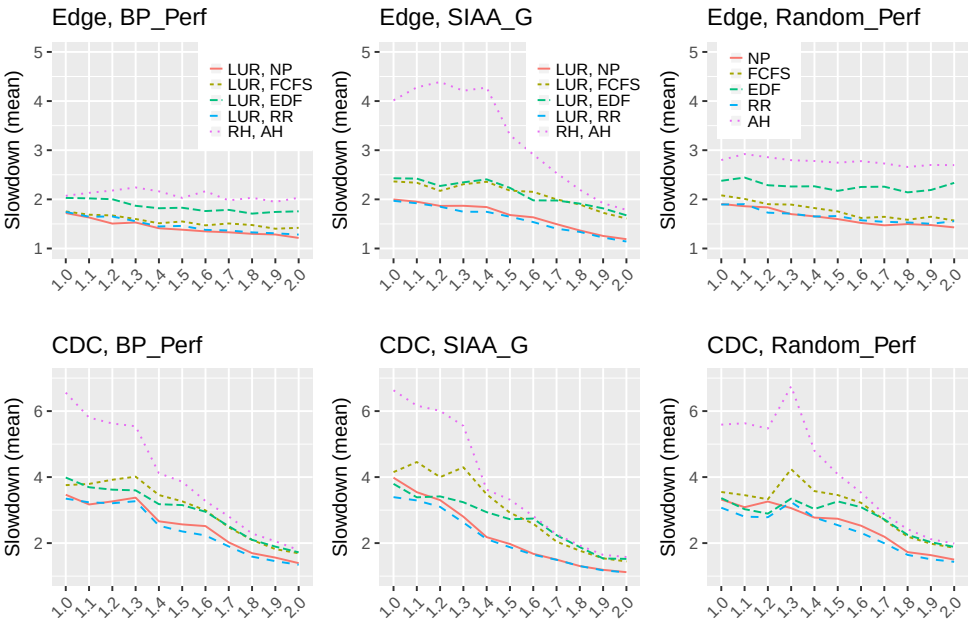

Figure 11: Mean slowdown as node speed increases ($\gamma = 0.1$)

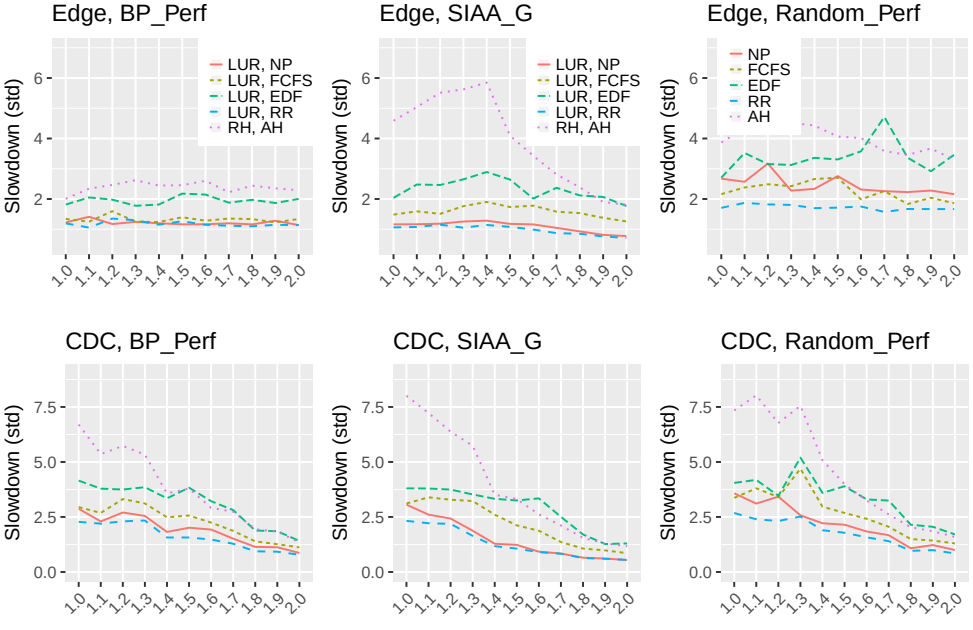

Figure 12: Standard deviation of slowdown as node speed increases ($\gamma = 0.1$)

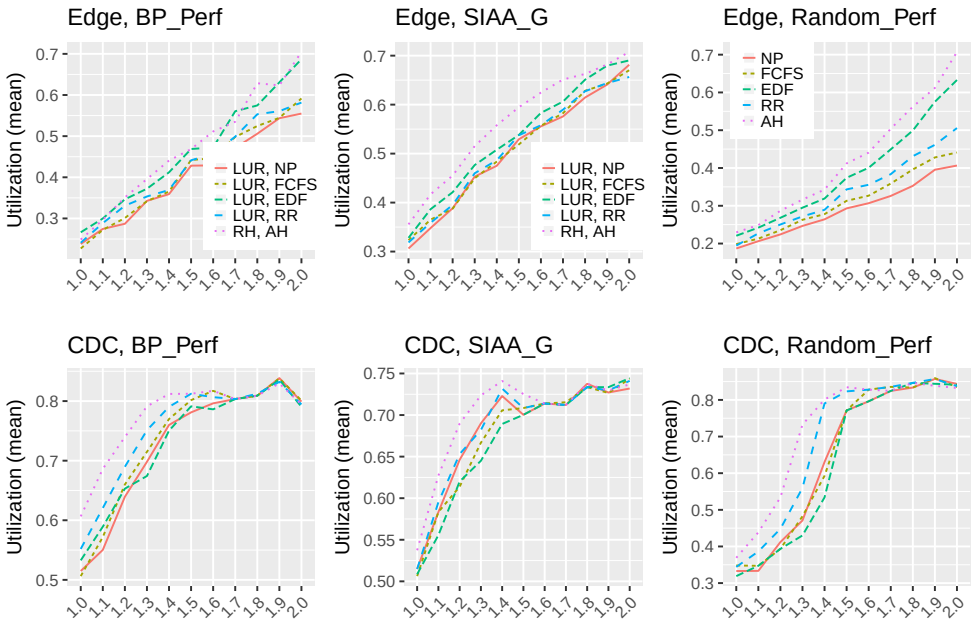

Figure 13: Mean utilization of runtimes as node speed increases ($\gamma = 0.1$)

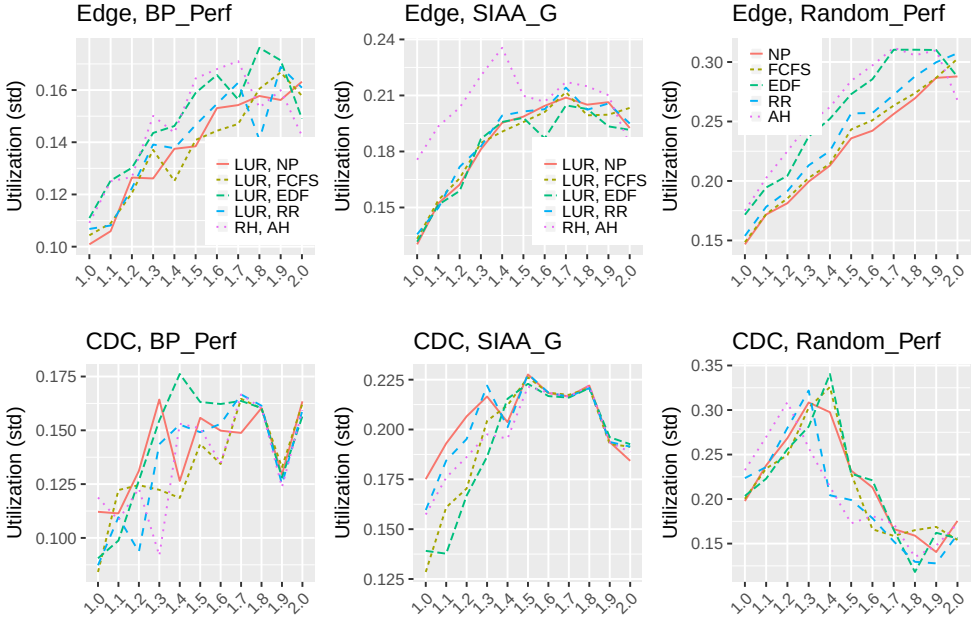

Figure 14: Standard deviation of utilization of runtimes as node speed increases ($\gamma = 0.1$)

largest decrease is for the combined SIAA_G_RH and AH policy, where both the average and standard deviation of application slowdown decreases by as much as 25% and 70%, respectively, when Edge capacity is increased from 15 to 1500 servers.

Both the average and standard deviation of the runtime utilization decreased for the Edge DC as the edge capacity increased, whereas it increased for the CDC. The largest decrease is for the combined SIAA_G_RH and the AH policy, where the average runtime utilization decreased by only 2% at the Edge DC and increased by 7% at the CDC.

Summary The combined policy SIAA_G_RH and AH – that is, one based on *runtime and application happiness* – performed better in terms of the number of runtimes required to place all the applications in the MEC compared to utilization-based policies. This policy allocates applications to runtime resources in such a way that the performance requirements of all applications are met. This differentiation among the applications can be seen in the slowdown metrics, where the results for the SIAA_G_RH and AH policy had highest average and standard deviation. The average and standard deviation for runtime utilization is also highest for this policy.

Size-based policies, such as SIAA_G_RH and SIAA_G_LUR, are based on the principle of hosting applications on the same runtime within a given service time range in order to decrease the service time variability on each runtime. This approach means that applications can be packed better than simple bin packing or random strategies. In general, the higher the variability of the service time of applications, the greater the gain from using size-based policies.

Increasing the node speed and edge capacity results in more applications being allocated to runtimes at the Edge DC. Increasing the node speed by 40% compared to the IoT devices minimized the number of runtimes required for application placement in the scenarios we evaluated.

7 Related Work

Scheduling has been studied by different communities. Hamadoui and Ramanathan investigated dynamic scheduling techniques that fulfill processing deadlines [25]. The works in [26, 27] explore scheduling of data stream processing (DSP). Cardellini *et al.* [28, 27] studied the joint optimization of operators replication and placement of DSP applications in geo-distributed environments. Mehta and Elmroth investigated distributed approach for allocation of applications in MECs by minimizing the system costs while meeting the performance requirements [1]. Tärneberg *et al.* [29] proposed a tractable locally optimal algorithm for dynamic application placement in MECs.

Categorization of applications into different QoS classes has been studied under different disciplines such as processor sharing [30], packet-switched networks [31],

storage systems [32], web servers [33] and cloud environments [7, 8]. Such work tries to ensure the performance targets of higher priority applications are met by taking resources from low-priority applications, with the implicit assumption that the spike in resource demand is short-lived and the low-priority applications will eventually be allocated sufficient resources to ensure their performance targets are also met. However, our solution works for both short-lived and long-lived resource shortage using the low-level and high-level schedulers, respectively. The low-level scheduler ensures the targets of critical applications are met by taking resources from low-priority applications to tolerate short-lived load spikes, whereas the high-level scheduler addresses persistent violations by migrating offending applications to a remote site or to less loaded nodes.

Moreover, most scheduling decisions are based on resource utilization [6, 8]. Such decisions are oblivious to the performance (targets) of applications and there is no guarantee that they will address or resolve performance issues. In contrast, our scheduler's decisions are informed by the happiness of an application and the runtimes. Happiness is an abstraction that captures the level of resources required for an application to perform as required (to be "happy"). In short, our approach ensures that the performance of applications is at the heart of the scheduler's decision-making process.

8 Conclusion

In this work, we analyze the performance of hierarchical schedulers for the placement of IIoT applications in a 5G and MEC environment. We propose a two-level scheduler: a high-level scheduler based on *runtime happiness* metrics and a low-level scheduler based on *application happiness* metrics. These metrics provide abstractions capturing the satisfaction of runtimes and hosted applications, relative to the performance targets of the applications. Our happiness-based, two-level scheduler is an alternative to schedulers based on runtime utilization metrics.

We evaluated combinations of four high-level policies and five low-level polices for the allocation of applications in the MEC for three scenarios: service time variation, MEC node speed variation and Edge DC capacity variation. Our results show that schedulers based on runtime happiness and application happiness outperformed those based on runtime utilization, including bin packing and random strategies. The combined policy SIAA_G_RH and AH required up to 30% fewer runtimes than policies based on runtime utilization in the scenarios we considered. This policy also increased runtime utilization by as much as 40% for the Edge DC used in our evaluation. Hence, we believe that *happiness*-based metrics provide a useful method for allocating applications (that have associated performance targets) to appropriate resources, and can be used to supplement or even replace utilization-based metrics.

Acknowledgment

Financial support has been provided by the Swedish Government's strategic research program eSSENCE and the Swedish Research Council project Cloud Control (C0590801).

References

[1] Amardeep Mehta and Erik Elmroth. Distributed Cost-Optimized Placement for Latency-Critical Applications in Heterogeneous Environments. In *Proceedings of the fifteenth IEEE International Conference on Autonomic Computing (ICAC)*, 2018.

[2] 5G-PPP, 5G Automotive Vision, white paper. `https://5g-ppp.eu/white-papers/`.

[3] Amardeep Mehta, Rami Baddour, Fredrik Svensson, Harald Gustafsson, and Erik Elmroth. Calvin Constrained - A Framework for IoT Applications in Heterogeneous Environments. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073. IEEE, 2017.

[4] Amardeep Mehta, William Tärneberg, Cristian Klein, Johan Tordsson, Maria Kihl, and Erik Elmroth. How Beneficial Are Intermediate Layer Data Centers in Mobile Edge Networks? In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 222–229, September 2016.

[5] Ewnetu Bayuh Lakew. *Autonomous cloud resource provisioning: accounting, allocation, and performance control*. PhD thesis, Umeå University, 2015.

[6] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 149192912X, 9781491929124.

[7] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-rodriguez, and Erik Elmroth. Performance-Based Service Differentiation in Clouds. In *15th IEEE/ACM International, Symposium on Cluster Computing and the Grid (CCGrid '15)*, pages 505–514, 2015.

[8] Luis Tomás, Ewnetu Bayuh Lakew, and Erik Elmroth. Service level and performance aware dynamic resource allocation in overbooked data centers. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 42–51, May 2016.

[9] Harshad Kasture and Daniel Sanchez. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[10] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, et al. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 14. ACM, 2017.

[11] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 15:1–15:13, New York, NY, USA, 2017. ACM.

[12] Johan Eker and Jorn W. Janneck. Dataflow programming in cal, 2014;balancing expressiveness, analyzability, and implementability. In *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 1120–1124, Nov 2012.

[13] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. `https://arxiv.org/abs/1008.1459`.

[14] J. Paul Morrison. *Flow-Based Programming, 2Nd Edition: A New Approach to Application Development*. CreateSpace, Paramount, CA, 2010.

[15] AWS Regions and Endpoints - Amazon Web Services. `https://docs.aws.amazon.com/general/latest/gr/rande.html`, .

[16] AWS Lambda – Pricing. `https://aws.amazon.com/lambda/pricing/`, .

[17] David Clark. A simple cost model for broadband access: What will video cost? In *Telecommunications Policy Research Conference. http://cfp. mit. edu/publications/docs/DDC. Cost. analysis. TPRC*, volume 1, 2008.

[18] What is a fair price for Internet service? - The Globe and Mail. `https://www.theglobeandmail.com/technology/gadgets-and-gear/what-is-a-fair-price-for-internet-service/article622177/`.

[19] How Much Does Data Really Cost an ISP? - Broadband Now. `https://broadbandnow.com/report/much-data-really-cost-isps/`.

[20] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[21] 5G specs announced: 20Gbps download, 1ms latency, 1M devices per square km | Ars Technica. https://arstechnica.com/information-technology/2017/02/5g-imt-2020-specs/.

[22] Matteo Fiorani, Paolo Monti, Björn Skubic, Jonas Mårtensson, Luca Valcarenghi, Piero Castoldi, and Lena Wosinska. Challenges for 5G transport networks. In *2014 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*, pages 1–6, Dec 2014.

[23] Kiichi Tateishi, Daisuke Kurita, Atsushi Harada, Yoshihisa Kishiyama, Shoji Itoh, Hideshi Murai, Stefan Parkvall, Johan Furuskog, and Peter Naucler. 5G Experimental Trial Achieving over 20 Gbps Using Advanced Multi-Antenna Solutions. In *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, pages 1–5, Sept 2016.

[24] Per Skarin, William Tärneberg, Karl-Erik Årzen, and Maria Kihl. Towards Mission-Critical Control at the Edge and Over 5G. *arXiv preprint arXiv:1803.02123*, 2018.

[25] Moncef Hamdaoui and Parameswaran Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE transactions on Computers*, 44(12):1443–1451, 1995.

[26] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 69–80, New York, NY, USA, 2016. ACM.

[27] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9):e4334, 2018.

[28] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, 87:171 – 185, 2018.

[29] William Tärneberg, Amardeep Mehta, Eddie Wadbro, Johan Tordsson, Johan Eker, Maria Kihl, and Erik Elmroth. Dynamic application placement in the Mobile Cloud Network. *Future Generation Computer Systems*, 70:163–177, May 2017.

[30] Leonard Kleinrock. Time-shared systems: a theoretical treatment. *J. ACM*, 14 (2):242–261, April 1967. ISSN 0004-5411.

[31] Navrati Saxena, Kalyan Basu, Sajal K. Das, and Cristina M. Pinotti. A new service classification strategy in hybrid scheduling to support differentiated QoS in wireless data networks. In *ICPP*, pages 389–396, 2005.

[32] Michael Mesnier et al. Differentiated storage services. In *Symposium on Operating Systems Principles (SOSP)*, pages 57–70. ACM, 2011. ISBN 978-1-4503-0977-6.

[33] Ying-Dar Lin et al. Multiple-resource request scheduling for differentiated QoS at website gateway. *Comput. Commun.*, 31(10), June 2008. ISSN 0140-3664.