URMILA: Dynamically Trading-off Fog and Edge Resources for Performance and Mobility-Aware IoT Services

Shashank Shekhar[a,1], Ajay Chhokra[b], Hongyang Sun[b], Aniruddha Gokhale[b,*], Abhishek Dubey[b], Xenofon Koutsoukos[b], Gabor Karsai[b]

[a] *Siemens Corporate Technology, Princeton, NJ 08540, USA.*
[b] *Vanderbilt University, Nashville, TN 37235, USA.*

Abstract

The fog/edge computing paradigm is increasingly being adopted to support a range of latency-sensitive IoT services due to its ability to assure the latency requirements of these services while supporting the elastic properties of cloud computing. IoT services that cater to user mobility, however, face a number of challenges in this context. First, since user mobility can incur wireless connectivity issues, executing these services entirely on edge resources, such as smartphones, will result in a rapid drain in the battery charge. In contrast, executing these services entirely on fog resources, such as cloudlets or micro data centers, will incur higher communication costs and increased latencies in the face of fluctuating wireless connectivity and signal strength. Second, a high degree of multi-tenancy on fog resources involving different IoT services can lead to performance interference issues due to resource contention. In order to address these challenges, this paper describes URMILA, which makes dynamic resource management decisions to achieve effective trade-offs between using the fog and edge resources yet ensuring that the latency requirements of the IoT services are met. We evaluate URMILA's capabilities in the context of a real-world use case on an emulated but realistic IoT testbed.

Keywords:
Fog/Edge Computing User Mobility Latency-sensitive IoT Services Resource Management

1. Introduction

Traditional cloud computing is proving to be inadequate to host latency-sensitive Internet of Things (IoT) applications due both to the possibility of violating their quality of service (QoS) constraints (e.g., due to the long round-trip latencies to reach the distant cloud) and the resource constraints (e.g., scarce battery power that drains due to the communication overhead and fluctuating connectivity). The fog/edge computing paradigm [1] addresses these concerns, where IoT application computations are performed at either the edge layer (e.g., smartphones and wearables) or the fog layer (e.g., micro data centers or cloudlets, which are a collection of a small set of server machines used to host computations from nearby clients), or both. The fog layer is effectively a miniaturized data center and hence supports multi-tenancy and elasticity, however, at a limited scale and with significantly less variety.

Despite the promise of fog/edge computing, many challenges remain unresolved. For instance, IoT applications tend to involve sensing and processing of information collected from one or more sources in real-time, and in turn making decisions to satisfy the needs of the applications, e.g., in smart transportation to alert drivers of congestion and take alternate routes. Processing this information requires sufficient computational capabilities. Thus, relying exclusively on edge resources alone for these computations may not always be feasible because one or both of the computational and storage requirements of the involved data may exceed the edge device's resource capacity. Even if it were feasible, the battery power constraints of the edge device limit how

*Corresponding Author.
Email addresses: shashankshekhar@siemens.com (Shashank Shekhar), ajay.d.chhokra@vanderbilt.edu (Ajay Chhokra), hongyang.sun@vanderbilt.edu (Hongyang Sun), a.gokhale@vanderbilt.edu (Aniruddha Gokhale), abhishek.dubey@vanderbilt.edu (Abhishek Dubey), xenofon.koutsoukos@vanderbilt.edu (Xenofon Koutsoukos), gabor.karsai@vanderbilt.edu (Gabor Karsai)

[1] Work performed by the author during doctoral studies at Vanderbilt University.

much intensive and for how long such computations can be carried out. In contrast, exclusive use of cyberforaging, i.e., always offloading the computations to the fog layer is not a solution either because offloading of data incurs communication costs, and when users of the IoT application are mobile, it is possible that the user may lose connectivity to a fog resource and/or may need to frequently hand-off the session between fog resources. In addition, the closest fog resource to the user may not have enough capacity to host the IoT application because other IoT applications may already be running at that fog resource, which will lead to severe performance interference problems [2, 3, 4, 5] and hence degradation in QoS for all the fog-hosted applications.

In summary, although the need to use fog/edge resources for latency-sensitive IoT applications is well-understood [6, 7], a solution that relies exclusively on a fog or edge resource is unlikely to deliver the desired QoS of the IoT applications, maintain service availability, minimize the deployment costs and ensure longevity of scarce edge resources, such as battery. These requirements are collectively referred to as the service level objectives (SLOs) of the IoT application. Thus, an approach that can intelligently switch between fog and edge resources while also supporting user mobility is needed to meet the SLO by accounting for latency variations due to mobility and execution time variations due to performance interference from co-located applications. To that end, we present *URMILA (Ubiquitous Resource Management for Interference and Latency-Aware services)*, which is a middleware solution to manage the resources across the cloud, fog and edge spectrum[2] and to ensure that SLO violations are minimized for latency-sensitive IoT applications, particularly those that are utilized in mobile environments. Specifically, this paper significantly extends our earlier work on URMILA [9] and makes the following key contributions:

- We provide an *a priori* estimate of the received signal strength that is then used at runtime to predict the energy consumption and network latency in the mobile environment by choosing an appropriate computing resource, i.e., edge or fog device.

- We formulate an optimization problem that minimizes the cost to the fog provider and energy consumption on edge devices while adhering to SLO requirements.

- We propose an algorithm to select the most suitable fog server that will be used to execute the IoT application remotely, when the computation can be executed on the fog resource. The algorithm accounts for performance interference due to co-located but competing IoT applications on multi-tenant fog servers and deliver a run-time control algorithm for application execution that ensures SLOs are met in real time.

- We evaluate our solution in a laboratory-sized real testbed using two emulated real-world IoT applications that we developed.

The rest of this paper is organized as follows: Section 2 discusses the application and the system models; Section 3 formulates the optimization problem and describes the challenges we address. Section 4 explains the URMILA solution in detail; Section 5 provides empirical validation of our work; Section 6 describes related work in comparison to URMILA; and finally Section 7 provides concluding remarks.

2. System Model and Assumptions

This section presents the system and application models for this research along with the assumptions we made.

2.1. System Model

Figure 1 is representative of a setup that our system infrastructure uses, which comprises a collection of distributed wireless access points (WAPs). WAPs leverage micro data centers (MDCs), which are fog resources. URMILA maintains a local manager at each MDC, and they all coordinate their actions with a global, centralized manager. The WAPs are interconnected via wide area network (WAN) links and hence may incur variable latencies and multiple hops to reach each other. The mobile edge devices have standard 2.4 GHz WiFi adapters to connect to the WAPs and implement well-established mechanisms to hand-off from one WAP to another. The edge devices are also provisioned with client-side URMILA middleware components including a local controller. We assume that mobile clients do not use cellular networks for the data transmission needs due to the higher monetary cost of cellular services and the higher energy consumption of cellular over wireless networks [10, 11].

[2]The use of the terms fog and edge, and their semantics are based on [8].

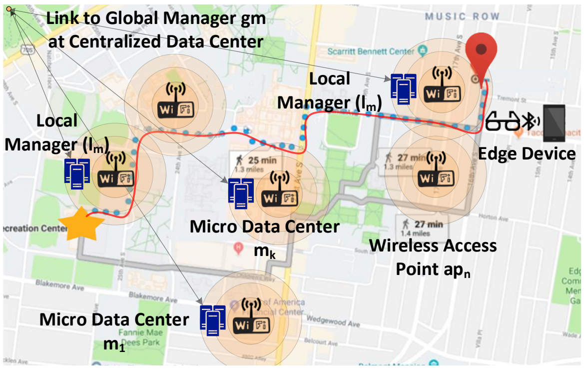


Figure 1: System infrastructure model

2.2. *Application Model*

We describe our IoT application model via a use case, which comprises a soft real-time object detection, cognitive navigational assistance application targeted towards the visually impaired. Advances in wearable devices and computer vision algorithms have enabled cognitive assistance and augmented reality applications to become a reality, e.g., Microsoft's SeeingAI (`www.microsoft.com/en-us/seeing-ai`) and Gabriel [1] that leverage Google Glass and cloudlets. However, because these solutions are either still not available to the users or use discontinued technologies such as Google Glass, we have developed two applications, which are also used in empirically validating our research and described in Section 5.1. As the user moves, the application frequently captures video frames of the surroundings using the wearable equipment, processes and analyzes these frames, and subsequently provides feedback (e.g., audio and haptics) to the user in real-time to ensure safe navigation. Note that our objective is not to replace service dogs or white canes but to augment the user's understanding of the surroundings.

Our use case belongs to a class of latency-sensitive IoT applications that are interactive or streaming in nature, such as augmented reality, online gaming, and cognitive assistance applications. The service level objective (SLO) for the service comprises multiple parts. First, since quality of user experience is critical, feedbacks are needed in (soft) real-time and hence we have tight deadlines for each step. Our application is modeled as a composition of individual tasks or steps; for instance, in the case of computer vision applications, these steps can be frame capturing, frame processing and actuation actions.

Since image processing is a compute- and memory-intensive application, it consumes the already scarce battery resources on a mobile device and hence the longevity of resources on edge devices is paramount. Although cyber-foraging enables a mobile application to be offloaded from the edge device to a fog/cloud node where it gets deployed and processed [12], this process itself is energy consuming because application state and logic needs to be transferred, and moreover it can be a platform-dependent issue, e.g., application binaries on different platforms may be different. Hence, in this work, we consider an approach where we have different versions of the service: one that can be deployed in containerized form at the fog node and another that runs on the edge device, albeit a less accurate but more resource efficient, so the service execution can switch between these two modes in order to maintain a highly available service and to meet the SLOs.

2.3. *User Mobility and Client Session*

To make effective resource management decisions, URMILA must estimate user mobility patterns. Although there exist both probabilistic and deterministic user mobility estimation techniques, for this research we focus on the deterministic approach, where the source and destination are known (e.g., via calendar events) or provided by the user *a priori*. Our solution can then determine a fixed route (or alternate sets of routes) for a given pair of start and end locations by leveraging external services such as Open Street Maps (`http://www.openstreetmap.org`), Here APIs (`https://developer.here.com/`) or Google Maps APIs (`https://cloud.google.com/maps-platform/`). These are reasonable assumptions for services like navigational assistance to the visually impaired or for students in or near college campuses who are using mobility-aware IoT applications where user mobility is restricted to a relative small geographical area, e.g., a couple of miles of user movement. Our future work will explore the probabilistic approach. When a user wants to use the application, a session is initiated, and the client-side application uses a RESTful API to inform URMILA about the start time, source and destination for the trip.

3. URMILA Problem Formulation

This section presents a formal description of the problem we solve in this paper. The aim is to meet the SLO for the user (which includes assuring the response time and minimizing the energy costs for the edge device by ensuring longevity of resources such as battery power) while minimizing the deployment and operational costs for the service provider. The primary notations we have used in the description are summarized in Table 1.

Table 1: Primary Notations Used in Problem Formulation

For application execution of a user u at a period p	
$\phi(u)$	bound on total response time or length of period
$I(u) = \{1(u), \ldots, L(u)\}$	sequence of periods, where L is the number of periods in the user's path
$t_{total}(u, p)$	total response time
$t_{process}(u)$	local pre/post-processing time of application
$t_{execute}(u, p)$	general execution time of application
$t_{local}(u)$	execution time when application is run locally
$t_{network}(u, p)$	network latency
For MDCs, servers and wireless access points	
gm	global manager
lm	local manager
M	set of MDCs
s	a server in an MDC
$AP = \{ap_1, \ldots, ap_n\}$	set of wireless access points
ap_0	virtual access point when user has no connection
$ap(s)$	access point that hosts server s
$ap(u, p)$	access point user u connects to at period p
$t_{ap(s),s}$	server latency between $ap(s)$ and server s
t_{ap_i,ap_j} or $t_{ap(u,p),ap(s)}$	latency between ap_i or $ap(u, p)$ and ap_j or $ap(s)$
$t_{u,ap(u,p)}$	last-hop latency between user u and $ap(u, p)$
For deployments of user u's application and associated costs	
$x_{u,s} \in \{0, 1\}$	deployment variable of user u on server s
$y_{u,s,p} \in \{0, 1\}$	execution variable of user u on server s at period p
$t_{remote}(u, s, p)$	execution time of user u on server s at period p
$t_{network}(u, s, p)$	total latency of user u on server s at period p
$U(s)$	set of existing users on server s
$L_{\max}(s)$	maximum duration $U(s)$ will run on server s
$T_{deploy}(u, s)$	cost of deploying user u on server s
$T_{transfer}(u, s)$	cost of state transfer of user u on server s
$w(u, s)$	waiting time of user s when deployed on server s
$T_{user}(u, s)$	no. of local periods for deploying user u on server s
$\alpha(s)$	unit-time cost of powering on server s
$\beta(s)$	unit-time cost of transferring state to server s
$\kappa(u)$	per-period energy cost of local execution for user u
$C(u)$	total cost of deploying user u

3.1. Formal Notation for the System Parameters

For each user (or application[3]), u, let $\phi(u)$ denote the user-specific bound on the acceptable response time in each service period, which also defines the length of the period. For our consideration, the total response time experienced by the user at each period p can be expressed as the sum of the (local or remote) execution time and the network latency (if executed remotely), i.e.,

$$t_{total}(u, p) = t_{process}(u) + t_{execute}(u, p) + t_{network}(u, p) \quad (1)$$

where $t_{process}(u)$ is the required total time of all the tasks associated with the application running locally. This duration is fixed and independent of the execution mode and period, $t_{execute}(u, p)$ is the total execution time of all the compute intensive tasks related to the application that can be offloaded to the remote server. This duration depends on whether the execution is on-device or remote, and $t_{network}(u, p)$ is the network latency for period p (which is included only if remote execution is involved). In the rest of the paper, $t_{execute}$ is referred to as the execution time of the application and $t_{process}$ as pre/post processing time of the application.

The goal is to meet the SLO for the user, i.e., to ensure $t_{total}(u, p) \leq \phi(u)$ for each period p in the user's anticipated duration of application usage, while minimizing the total cost (formulated in Section 3.2). Since we consider user mobility, this duration is typically from the start to the end of the user's trip. Nonetheless, there is nothing to prevent us from applying the model even in the stationary state or after the user has reached his/her destination.

Let $t_{local}(u)$ denote the execution time when the application of user u is run locally, which is fixed regardless of the period and no network latency will be incurred in this case. Additionally, we assume that the SLO can always be satisfied with local executions, i.e., $t_{process}(u) + t_{local}(u) \leq \phi(u)$ for all u and p. This could be achieved by a lightweight mobile version of the application, such as MobileNet for real-time object detection on the mobile device, which is less compute-intensive and time-consuming, thereby ensuring the SLO albeit with a low detection accuracy.

In our model, applications and fog resources are managed by a centralized authority known as the global manager (gm) hosted at a centralized cloud data center (CDC). This serves as URMILA's portal for the users. We denote by $AP = \{ap_1, ap_2, \ldots, ap_n\}$ the set of Wireless Access Points (WAPs) with a subset of them also hosting fog resources in the form of micro data centers (MDCs) or cloudlets. A WAP, $ap \in AP$, hosting an MDC, $m \in M$, implies that the access point ap is directly connected to wired local area network involving all the servers of m. Such capabilities could be offered by college campuses or internet providers as wireless hotspots. We assume that the gm owns or has exclusive lease to a set M of MDCs. Note that M is a subset of AP since only some WAPs have an associated MDC. Each MDC contains a set of compute servers (possibly heterogeneous) that are connected to their MDC's associated WAP. From a traditional cloud computing perspective, since an application can be deployed and executed on the CDC, we model the CDC as a special MDC that is also contained in set M, and correspondingly, the set AP contains the access point that hosts the CDC as well.

In this architecture, the network latency between any $ap(s) \in AP$ that hosts a server s and the server itself is negligible, i.e., $t_{ap(s),s} \approx 0$, as they are connected via fast local area network (LAN). The WAPs are connected to each other over a wide area network (WAN) and may

[3]Since the mobile user is engaged using the features of a single application, we will use the terms "user" and "application" interchangeably.

incur significant latency depending on the distance, connection type and number of hops between them. If a mobile user is connected to a nearby WAP, say ap_i, which has an MDC that hosts the user's application, then there is no additional access point involved. Otherwise, if the application is deployed on another MDC hosted by, say ap_j, then the round trip latency t_{ap_i,ap_j} can be significant since the request/response will be forwarded from ap_i to ap_j. Moreover, due to mobility, the user could at times have no connection to any access point (e.g., out of range). In this case, we assume the presence of a virtual access point ap_0 to which the user is connected and define $t_{ap_0,ap_i} = \infty$ for any $ap_i \in AP$. Obviously, the application will have to run locally to avoid SLO violations.

In addition to the round trip latency, the selection of MDC and server to deploy the application can also significantly impact the application execution time, since the MDCs can have heterogeneous configurations and each server can host multiple virtualized services, which do not have perfect isolation and hence could interfere with each other's performance.Each MDC, also contains a local manager lm responsible for maintaining a database of applications it can host, their network latencies for the typical load, and server type and load-specific application execution time models. Note that there could be a varying number of co-located applications and hence a varying load on each server over time, but we assume that individual application's workload does not experience significant variation throughout its lifetime, which is a reasonable assumption for many streaming applications, such as processing constant size video frames.

For our mobility model, we divide the travel duration for each user u into a sequence $I(u) = \{1(u), 2(u), \ldots, L(u)\}$ of periods that cover the user's path of travel. The length of each period $p \in I(u)$ is the same and sufficiently small so that the user is considered to be constantly and stably connected to a particular WAP $ap(u, p) \in AP \bigcup \{ap_0\}$ (including the virtual access point). Moreover, the last hop latency, $t_{u,ap(u,p)}$ between the user and this access point can be estimated based on the user's position, channel utilization, and number of active users connected to that access point.

3.2. Developing the Problem Statement

To formalize the optimization problem we solve in this work, we define two binary variables that indicate the decisions for application deployment and execution mode selection. Specifically, $x_{u,s} = 1$ if user u is deployed on server s and 0 otherwise, and $y_{u,s,p} = 1$ if user u executes on server s at period p and 0 otherwise. Using these two variables and our system model, we now express the total response time of an application and the total cost, and then present the complete formulation of the optimization problem.

3.2.1. Characterizing the Total Response Time

Recall from Equation (1) that the total response time for a user u at a period p consists of three parts, and among them the pre/post-processing time $t_{process}(u)$ is fixed. To express the execution time, let $t_{remote}(u, s, p)$ denote user u's execution time if it is run remotely on server s at period p. Note that, due to the hardware heterogeneity and co-location of multiple applications on the server which can result in performance interference [13, 4, 14], this execution time will depend on the set of existing applications that are running on the server at the same time. This property is known as *sensitivity* [15, 13, 3]. Similarly, the execution times for these users may in turn be affected by the application execution of user u were it to execute on this server – a property known as *pressure* [15, 13, 3]. Techniques to estimate $t_{remote}(u, s, p)$ are described in Section 4.4.

For the network latency, let $t_{network}(u, s, p)$ denote the total latency incurred by running the application remotely on server s at period p. We can express it as:

$$t_{network}(u, s, p) = t_{u,ap(u,p)} + t_{ap(u,p),ap(s)} + t_{ap(s),s} \qquad (2)$$

In particular, it includes the latency from the user to the connected access point $t_{u,ap(u,p)}$, which we refer to as the *last-hop latency*; the latency from the connected access point to the serving access point $t_{ap(u,p),ap(s)}$, which we refer to as the *WAN* latency; and the latency from the serving access point to the server that deploys the application $t_{ap(s),s}$, which we refer to as the *server latency*. The last latency is negligible, and the first two depend on the user's location at period p. Latency estimation is discussed in Section 4.3. The total response time of user u at period p can then be expressed as:

$$\begin{aligned} t_{total}(u, p) = t_{process}(u) + \Big(1 - \sum_s y_{u,s,p}\Big) t_{local}(u) \\ + \sum_s y_{u,s,p}\Big(t_{remote}(u, s, p) + t_{network}(u, s, p)\Big) \end{aligned} \qquad (3)$$

In the above expression, the first line includes the constant pre/post-processing time as well as the execution time when the application runs locally, and the second line includes the execution time when it is run remotely as well as the incurred total network latency.

3.2.2. Characterizing the Total Cost

The total cost consists of two parts: the server deployment cost and the user energy cost. On the deployment side, running a server incurs operational costs, such as power and cooling. Thus, the provider want to use as few server-seconds as possible, so the deployment cost depends on the duration a server remains running. For a server s, let $U(s)$ denote the set of existing users whose applications are deployed on it, and the maximum time up to which a server will run these applications depends upon the longest running application, i.e., $L(v)$, where L is the number of periods in the user v's path. We define $L_{\max}(s)$ to be the maximum time up to which these existing applications will run, i.e., $L_{\max}(s) = \max_{v \in U(s)} L(v)$. The cost for deploying a new application u on server s is proportional to the extra duration the server has to be on and can be expressed as:

$$T_{deploy}(u, s) = \max\left(0, L(u) - L_{\max}(s)\right) \tag{4}$$

In addition to the operational cost, deploying an application on a server requires transferring its state over the backhaul network from the repository in the CDC to the MDC. The time to transfer the state of an application u to a server s can be expressed as:

$$T_{transfer}(u, s) = \frac{state(u)}{b(s)} + ci(u, s) \tag{5}$$

where $state(u)$ is the size of application u's state, $b(s)$ is the backhaul bandwidth from CDC to the MDC that hosts server s, and $ci(u, s)$ is the initialization time of the application before it can start processing requests on the server. Hence, the waiting time (in terms of the number of periods) of the application before it can be executed remotely is $w(u, s) = \left\lceil T_{transfer}(u, s)/\phi(u) \right\rceil$, where $\phi(u)$ is the duration of a period. Thus, we must have $y_{u,s,p} = 0$ for $p \in [1(u), 1(u) + w(u, s)]$.

On the user side, we know that executing the application locally incurs higher power consumption than executing it remotely. Hence, the cost for user u can be measured in terms of the total number of periods when the application is being run locally, which is directly proportional to the additional energy expended by the mobile device had the application been run remotely throughout the user's travel. The number of local periods by deploying application u on server s can be expressed as:

$$T_{user}(u, s) = \sum_{p=1(u)}^{L(u)} \left(1 - y_{u,s,p}\right) \tag{6}$$

To combine the costs from different sources, we define $\alpha(s)$ and $\beta(s)$ to be the unit-time costs of powering on server s and transferring the state to server s, respectively. Both values depend on the server and its corresponding MDC. In addition, we define $\kappa(u)$ to be the per-period energy cost of local execution for user u (relative to remote executions), and its value depends on the user's application and mobile device. Thus, for a given solution that specifies the application deployment (i.e., $x_{u,s}$) and its execution mode for each period (i.e., $y_{u,s,p}$), the total cost can be expressed as:

$$C(u) = \sum_{s} x_{u,s}\Big(\alpha(s) \cdot T_{deploy}(u, s) + \beta(s) \cdot T_{transfer}(u, s) + \kappa(u) \cdot T_{user}(u, s)\Big) \tag{7}$$

3.3. Optimization Problem

Given the expressions for total response time (Equation (3)) and total cost (Equation (7)), the optimization problem needs to decide, for each new user u, where to deploy the application and which execution mode to run the application in order to minimize the total cost subject to the response time constraints. Let V denote the set of all existing applications on all servers at the time of deploying u, i.e., $V = \bigcup_s U(s)$. The problem can be formulated by the following integer nonlinear program (INLP):

$$\begin{aligned}
\text{minimize} \quad & C(u) \\
\text{subject to} \quad & t_{total}(u, p) \le \phi(u), \quad \forall p && (8) \\
& t_{total}(v, p) \le \phi(v), \quad \forall p, v && (9) \\
& x_{u,s}, y_{u,s,p} \in \{0, 1\}, \quad \forall s, p && (10) \\
& \sum_{s} x_{u,s} \le 1 && (11) \\
& y_{u,s,p} \le x_{u,s}, \quad \forall s, p && (12) \\
& y_{u,s,p} = 0, \quad \forall s, p \in [1(u), 1(u) + w(u, s)] && (13)
\end{aligned}$$

In particular, Constraints (8) and (9) require meeting the SLOs for user u as well as for all existing users at all times. Constraint (10) requires the decision variables to be binary. Constraint (11) requires the application to be deployed on at most one server. We enforce this constraint because there is a high cost in transferring the application state from the CDC to an MDC server, initializing and running it. Note that an application need not be deployed on any server, in which case it will be executed locally throughout the user's travel duration. Constraint (12) allows the application to run remotely only on the server it is deployed at each period and Constraint (13) restricts the remote executions to start only after the application state has been transferred.

Due to the NP-hardness of the above INLP problem, we rely on a greedy-based heuristic to solve it. Section 4.4.2 describes the proposed heuristic for server deployment and execution mode selection.

4. URMILA: Design and Implementation

This section presents the design and implementation of our URMILA dynamic resource management middleware.

4.1. Overview of URMILA's Expected Runtime Behavior

To better understand the rationale for URMILA's design and its architecture, let us consider the runtime interactions that ensue once a user session is initiated. The client-side application is assumed to be aware of URMILA and communicates with it to provide the start time, source and destination for the trip. URMILA computes the set of routes that the user may take using the provided trip details. Then, based on instantaneous loads on all fog nodes of the MDCs on the path, URMILA determines a suitable fog server (i.e., node) in an MDC on which the IoT application's cloud/fog-ready task can be executed throughout the session, and deploys the corresponding task on that server. URMILA will not change this selected server for the rest of the session even if the user may go out of wireless range from it because the user can still reach it through a nearby WAP and by traversing the WAN links. This is reasonable for our approach due to the relatively smaller size of the geographical area covered by the mobile user.

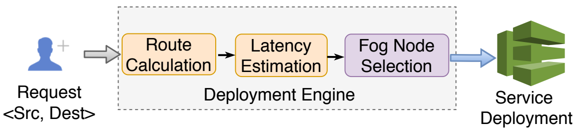


Figure 2: URMILA's Component-based Architecture and Deployment

This approach and the architectural components involved in the process are depicted in Figure 2. This sequence is repeated whenever a new user is added to the system. Selecting the appropriate fog server based on the instantaneous utilizations of the available resources, which are not known statically, while ensuring SLOs are met is a hard problem. URMILA's key contribution lies in addressing this challenge, and intelligently adapting between the fog and edge resources based on user mobility and application SLO.

As time progresses, for each period (or a well-defined epoch) of application execution, the client-side URMILA middleware determines the instantaneous network conditions and determines whether to process the request locally or remotely on the selected fog server such that the application's SLO is met. This process continues until the user reaches the destination and terminates the session with the service, at which point the provisioned tasks on the fog resources can be terminated. The architecture for these interactions is presented in Figure 3, where the controller component on the client-side middleware is informed by URMILA to opportunistically switch between fog-based or edge-based execution in a way that meets application SLOs. The remainder of this section describes how URMILA achieves these goals.

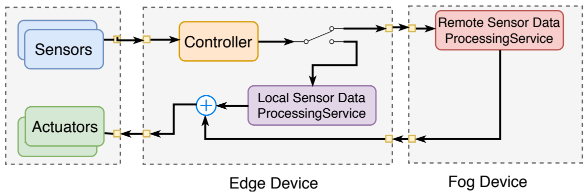


Figure 3: URMILA's Architecture for Decision Making

4.2. Route Computation

This component is responsible for determining the user's mobility pattern based on the methodology described in Section 2.3. In this paper, we leverage the Google Maps APIs for finding the shortest route between the user's specified start and destination locations. It takes a tuple comprising the start and destination GPS coordinates, and produces a list of GPS coordinates for the various steps along the route. This raw list of route points is re-sampled as per a constant velocity model (5 kilometers per hour, which is a typical average walking speed) with an interval equal to the response time deadline enforced by the SLO.

4.3. Latency Estimation

Recall that URMILA will choose to execute task(s) of the IoT application on the fog server if it can assure its SLO, which means that for every user and for every period/epoch of that user's session, URMILA must be able to estimate the expected latency as the user moves along the route. Hence, once the route (or set of alternate routes) taken by the user is determined using mechanisms like Google Maps, the *Latency Estimator* component of Figure 2 will estimate the expected latencies along the route.

This is a hard problem to address due to the dynamic nature of the Wi-Fi channels and the dynamically changing traffic patterns (due to changing user densities) throughout the day. To that end, URMILA employs a data-driven model that maps every route point on the path to an expected latency to be observed at that point. One of the salient features of this estimation model is its adaptability, i.e., the model is refined continuously in accordance with the actual observed latencies.

The estimated latency is made up of three parts (see Equation (2)): the last-hop latency to a WAP along the route, the WAN latencies to reach the fog server from the ingress WAP by traversing the WAN links, and the task execution time on the fog server (See Section 4.4).

Estimating Last-hop Latency $t_{u,ap(u,p)}$: The last hop latency itself is affected primarily by *channel utilization*, *number of active users* and *received signal strength* [16]. This component estimates the latency $t_{network}(u, s, p)$ observed by user u at any period p along the route on any given server s. Initially, we assume that the channel utilization and the number of active users do not impact the latency significantly. As the routes get profiled, we maintain a database that stores network latencies for different coordinates and times of the day. Whenever a request arrives with known route segments, the latency can be estimated by querying this database.

Equation (14) can be used to compute the signal strength, where $\hat{p}$ (resp. $\hat{p_0}$) is the mean received power at a distance d (resp. d_0) from the access point, and γ is the path loss exponent. Among these parameters, $\hat{p_0}$ and d_0 depend on the access point and are known *a priori* for typical access points. The path loss exponent γ depends on the environment, and its typical values for free space, urban area, sub urban area and indoor (line of sight) are 2, 2.7 to 3.5, 3 to 5 and 1.6 to 1.8, respectively [17].

$$\hat{p}(d) = \hat{p_0}(d_0) - 10\gamma \log \frac{d}{d_0} \tag{14}$$

The client device selects a WAP with the highest signal strength and sticks to it till the strength drops below a threshold. The network becomes unreliable if the received signal strength falls below -67dBm for streaming applications [16], which we use as the threshold for URMILA. We also use existing well-known methods for determining the signal strength based on received power and distance from an access point [17]. Using this together with the calculated route and WAP's data, the latency estimator is able to calculate the last-hop latency for each period/epoch along the route.

Estimating WAN Latency $t_{ap(u,p),ap(s)}$: The WAN latency between two access points depends on the *link capacity* connecting the nodes and the number of hops between them. We use another database to maintain the latencies between different access points.

Estimating Total Latency: Based on the computed individual components, a map of total network latency can then be generated for every period/epoch along the route.

4.4. Fog Server Selection

To avoid the high cost involved in transferring application state and initialization, URMILA performs a one-time fog server selection within a fog layer, and reserves the resource for the entire trip duration plus a margin to account for the deviation from the ideal mobility pattern. To determine the right fog server to execute the task, besides having accurate latency estimates, we also need an accurate estimate for task execution on the fog server that will end up being selected, which will depend on the instantaneous co-located workloads on that server and the incurred performance interference.

To accomplish this, we leverage the INDICES [7] performance metric collection and interference modeling framework. However, the INDICES framework has a few limitations. In particular, it was designed for virtual machines (VMs). In this work, in order to have lower initialization cost compared to VMs [18], we rely on Docker containers. Hence, as a part of URMILA, we integrated INDICES while extending the framework for interference-aware Docker container deployment.

In addition, modern hardwares are equipped with non uniform memory access (NUMA) architecture which forces the performance estimation and scheduling techniques to consider memory locality. Different applications have different levels of performance sensitivity on NUMA architectures [19]. Thus, we needed a mechanism that is able to benchmark applications on different NUMA nodes and predict their performance and schedule them accordingly. Moreover, recent advancement in Resource Director Technology (RDT) [20] that includes Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) provides further insights about system resource consumption for memory bandwidth and last-level cache utilization, which can be leveraged for better performance estimation. We account for all of these factors in URMILA. Our recent work on the FECBench framework addresses several of the limitations in INDICES and provides a holistic, end-to-end performance monitoring and model building framework [21], however, FECBench was not ready for use in the URMILA research.

URMILA's fog server selection process consists of an offline performance modeling stage and an online server selection stage as depicted in Figure 4.

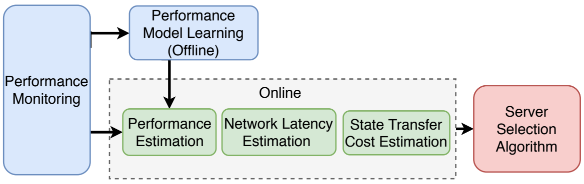


Figure 4: URMILA's Fog Server Selection Process

4.4.1. Offline Performance Model Learning

URMILA uses a data-driven approach [?] in its run-time decision making for which it requires an offline training stage to develop a performance model for each latency-sensitive application task that is expected to be executed on the fog server. More precisely, in order to calculate $t_{execute}(u, p)$ in Equation (1), we need to develop a performance model to predict $t_{remote}(u, s, p)$, the remote execution time of the application on a server. This model depends on the following two factors:

1. *Hardware Heterogeneity*: Our edge and fog resources are composed of heterogeneous hardware with different server architectures and configurations. Each application's performance can vary significantly from one platform to another [2]. Therefore, we need an accurate benchmark of performance for each hardware platform.
2. *Performance Interference*: Although hypervisors/virtual machine monitors and cgroups in case of Linux containers provide a high degree of security, fault, and environment isolations, there still exist a number of interference sources [13, 4, 14], such as shared last-level cache, interconnect network, disk and memory bandwidth which are difficult to partition. This has a profound impact on the remote execution time ($t_{remote}(u, s, p)$), arising from the sensitivity to the co-located applications and its pressure on those applications [15, 13, 3].

To develop a performance model required for determining $t_{remote}(u, s, p)$, we first benchmark the execution time $t_{isolation}(u, w)$ of each latency-sensitive application u on a specific hardware type w in isolation. This way, we can account for the hardware heterogeneity of our resource spectrum. We then execute the application with different co-located workload patterns and learn its impact, denoted by function g_u, on the system-level and obtain micro-architectural metrics as follows:

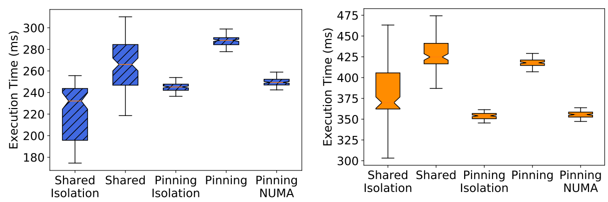
$$\mathbf{X}_w^{new} = g_u(\mathbf{X}_w^{cur}) \tag{15}$$

where $\mathbf{X}_w^{cur}$ and $\mathbf{X}_w^{new}$ denote the vectors of the selected metrics before and after running application u on hardware w, respectively.

Modern hardware architectures provide access to many performance metrics. Based on our sensitivity analysis and to provide a broadly applicable and easily reproducible approach, we selected the following metrics in vector $\mathbf{X}_w^{cur}$ for performance modeling:

- *System Metrics*: CPU utilization, memory utilization, network I/O, disk I/O, context switches and page faults.
- *Hardware Counters*: Retired instructions per second (IPS), cache utilization, cache misses, last-level cache (LLC) bandwidth and memory bandwidth.
- *Scheduler Metrics*: Scheduler wait time and scheduler I/O wait time.

Another key consideration that we applied for performance modeling is NUMA-awareness with CPU core pinning. On modern multi-chip servers, the memory is divided and configured locally for each processor. The memory access time is lower when accessed from local NUMA node compared to when accessed from remote NUMA node. Hence, it is desirable to model the performance per NUMA node and schedule the Docker containers accordingly. We achieve this by collecting the performance metrics per NUMA node and then, wherever possible, developing sensitivity and pressure profiles at the NUMA node level instead of at the system level. The benefit of this approach is validated in Figure 5. We observe from the figure that CPU core pinning reduces the performance variability, however, if NUMA node is not accounted for, it could lead to worse performance due to data locality issues.



(a) MobileNet (b) Inception

Figure 5: Execution Time Comparison due to Core Pinning and NUMA

Lastly, we learn the performance deterioration (compared to isolated performance), denoted by function f_u, for application u under the new metric vector $\mathbf{X}_w^{new}$ on hardware w to predict its execution time on the fog server under the same conditions:

$$t_{remote}(u, w) = t_{isolation}(u, w) \cdot f_u(\mathbf{X}_w^{new}) \quad (16)$$

We apply supervised machine learning techniques to learn both functions g_u and f_u using the following sequence of steps:

- *Feature Selection*: We adopted the Recursive Feature Elimination (RFE) approach using Gradient Boosted Regression Trees [22] as a way to select the optimal set of features and reduce training time. We performed RFE in a cross-validation loop to find the optimal number of features that minimizes a loss function (mean squared).
- *Correlation Analysis*: To further reduce the training time by decreasing the dimensions of the feature vector, we used the Pearson Coefficient to eliminate highly dependent metrics with a threshold of ±0.8.
- *Regression Analysis*: We used the off-the-shelf Gradient Tree Boosting curve fitting method due to its ability to handle heterogeneous features and its robustness to outliers.

Note that Equations (15) and (16) can be applied together to model both sensitivity and pressure for application deployment on each server in order to calculate $t_{remote}(u, w)$, which is then used as an estimate for the remote execution time $t_{remote}(u, s, p)$ of application u on server s containing hardware w. The learned performance models for different applications are then distributed to the different MDCs for each of the hardware type w that they contain. Since MDCs typically contain just a few heterogeneous server types, we do not anticipate a large amount of performance model dissemination.

4.4.2. Online Server Selection

The online stage performs server selection for an application, which is done in a hierarchical fashion as follows. First, when a user initiates a session, the global manager *gm* residing at the CDC initiates the fog server selection process as soon as it receives a request from the client application. It calculates the route of the user as described in Section 4.2. Recall that the goal is to determine the expected execution time of the application task on each fog server in the most appropriate MDC using the performance model developed in the offline stage such that the SLOs for the existing applications can still be met despite expected performance interference. Thus, once URMILA knows the route and the access points the user will be connected to, the *gm* then queries the local manager *lm* of each MDC, which in turn queries each of their servers to find the expected execution time of the target application using the performance model developed in the offline stage such that the SLOs for the existing applications can still be met. Finally, the *gm* combines this information with the latency estimates from Section 4.3 to determine the execution mode of the application to satisfy the response time constraints at each step of the route. This allows us to estimate the cost incurred by the user (i.e., T_{user} in Equation (6)).

To solve the optimization problem, we still need to estimate the deployment cost (i.e., T_{deploy} in Equation (4)) and the transfer cost (i.e., $T_{transfer}$ in Equation (5)). The deployment cost is based on the trip duration, which we can again obtain from the user mobility as described in Section 4.2. To reduce transfer cost, we use Docker container images that consist of layers, and each layer other than the last one is read only and is made of a set of differences from the layer below it. Thus, with a base image (such as Ubuntu 16.04) already present on the server, we only need the delta layers (that dictate $state(u)$ in Equation (5)) to be transferred for the application to be reconstructed at the fog location.

Algorithm 1 shows the pseudocode for selecting a fog server s^* and deciding a tentative execution-mode plan $y^*[p]$ for a user u at each period/epoch p in the route, where $y^*[p] = 1$ indicates remote execution and $y^*[p] = 0$ indicates local execution. Besides deciding on the server to deploy the target application, the algorithm also suggests a tentative execution-mode plan at each step of the application execution. This execution plan will be used for cost estimation by the global manager and is subject to dynamic adjustment at run-time (See Section 4.5).

Specifically, the algorithm goes through all servers (Line 3), and first checks whether deploying the target application u on a server s will result in SLO violation for each existing application v on that server, as specified by the user's response time bound $\phi(v)$ (Lines 4-15). For each application v, its total response time consists of a fixed pre-processing time $t_{process}$, an execution time and a network latency. Since it may have a variable network latency and a variable execution time depending on the user's location and choice of execution mode, we should ideally check for its SLO at each period of its execution. However, doing so may incur unnecessary overhead on the global manager since the execution-mode plan for v is also tentative. Instead, the algorithm

Algorithm 1: Fog Server Selection

Input: Target application u and other information on the user's route, networks, servers and their loads

Output: Server s^* to deploy u and a tentative execution mode vector $y^*[p] \in \{0, 1\}$ for each period p during the user's route

1 **begin**
2 Initialize $cost_{\min} \leftarrow \infty$, $s^* \leftarrow \emptyset$, and $y^*[p] \leftarrow 0\ \forall p$;
3 **for** each server s **do**
4 $\mathbf{X}^{cur} \leftarrow GetCurrentSystemMetrics(s)$;
5 $\mathbf{X}^{new} \leftarrow g_u(\mathbf{X}^{cur})$;
6 $V \leftarrow GetListOfExistingApplications(s)$;
7 **for** each application $v \in V$ **do**
8 $t_{process} \leftarrow GetPreProcessingTime(v)$;
9 $t_{isolation} \leftarrow GetIsolatedExecTime(v, s)$;
10 $t_{remote} \leftarrow t_{isolation} \cdot f_v(\mathbf{X}^{new})$;
11 $t^{SLO}_{network} \leftarrow GetPercentileLatency(v, s)$;
12 **if** $t_{process} + t_{remote} + t^{SLO}_{network} > \phi(v)$ **then**
13 **skip** s;
14 **end**
15 **end**
16 Initialize $y[p] \leftarrow 0\ \forall p$; // execute locally by default;
17 $t_{process} \leftarrow GetPreProcessingTime(u)$;
18 $t_{isolation} \leftarrow GetIsolatedExecTime(u, s)$;
19 $t_{remote} \leftarrow t_{isolation} \cdot f_u(\mathbf{X}^{new})$;
20 $T_{deploy} \leftarrow GetDeploymentCost(u, s)$;
21 $T_{transfer} \leftarrow GetStateTransferCost(u, s)$;
22 **for** each period p in the route **do**
23 $t^{SLO}_{network}(p) \leftarrow GetPercentileLatency(u, s, p)$;
24 **if** $t_{process} + t_{remote} + t^{SLO}_{network}(p) \leq \phi(u)$ **then**
25 $y[p] \leftarrow 1$; // execute this period remotely;
26 **end**
27 **end**
28 $T_{user} \leftarrow ComputeUserCost(y)$;
29 $cost \leftarrow \alpha \cdot T_{deploy} + \beta \cdot T_{transfer} + \kappa \cdot T_{user}$;
30 **if** $cost \leq cost_{\min}$ **then**
31 $cost_{\min} \leftarrow cost$;
32 $s^* \leftarrow s$ and $y^* \leftarrow y$;
33 **end**
34 **end**
35 **end**

considers the estimated network SLO percentile latency $t^{SLO}_{network}$ (e.g., 90^{th}, 95^{th}, 99^{th}) while assuming that in the worst case the application always executes remotely for the execution time, i.e., t_{remote}. This approach provides a more robust performance guarantee for existing applications in case of unexpected user mobility behavior.

Subsequently, for each feasible server, the algorithm evaluates the overall cost of deploying the target application u on that server (Lines 16-29) and chooses the one that results in the least cost (Lines 30-33). Note that the overall cost consists of the server deployment cost T_{deploy} and application state transfer cost $T_{transfer}$, both of which are fixed for a given server, as well as the user' energy cost T_{user}, which could vary depending on the execution mode vector y. Hence, to minimize the overall cost, the algorithm offloads the execution to the remote server as much as possible subject to its SLO being met (Lines 22-27).

4.5. RunTime Phase

The deployment phase outputs the network address of the fog server where the application will be deployed and a list of execution modes as shown in Algorithm 1. This information is relayed to the client-side middleware, which then starts forwarding the application data to the fog server as per the execution mode at every step. However, the execution mode list is based on the expected values of the network latencies, and hence can be different from the actual value.

The runtime phase minimizes the SLO violations due to inaccurate predictions by employing a robust mode selection strategy that updates the decision at any step based on the feedback from previous steps. As shown in Figure 3, the *Controller* obtains sensor data and selects appropriate mode for processing the data. The processed data is transformed and fed back to actuators which provides the user with output using the chosen medium (voice description of the classified object in our use case application).

The *Controller* consists of a process, *Mode Selector,* which is responsible for gathering sensor data, selecting appropriate mode and monitoring the timing deadline violations. *Mode Selector* is modeled using Mealy machine, M_{sel} as shown in Figure 6. M_{sel} consists of 7 symbolic states with `Idle` being the initial state. From `Idle` state, the state machine transitions to `SyncWithSLO` state after receiving *Start* event. The transition from `SyncWithSLO` is caused by the activation of $TimeOut(t_2)$ event that pushes the state machine into `GatheringSensorData` while emitting *GetSensorData* event. This event activates a system level process to pull data from various sensors. If this task is not completed in t_3 secs, the $TimeOut(t_3)$ event forces the state machine back to `SyncWithSLO`. If the task of acquiring sensor data finishes before deadline, the state machine transitions to `SelectingMode` while producing *EvaluateConn* event.

EvaluateConn starts another asynchronous process, p, to acquire signal strength level and check the estimated execution mode. If the execution mode is remote and signal strength is above the threshold, only then remote mode is selected at run time, which is signaled by this asynchronous task by emitting *SwitchToRemote* event, that enables M_{sel} to jump to `SendingData`. However, in the past if for the same access point, both the conditions were met and yet timing deadline had failed,

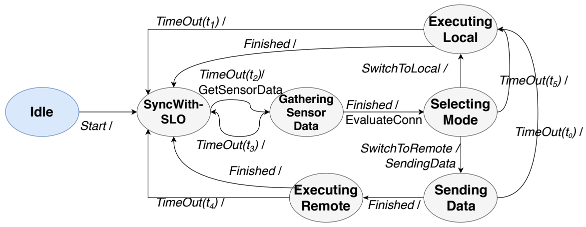


Figure 6: Mode Selector State Machine

then local mode will be selected as long as client device is connected to the same access point.

After getting *SwitchToRemote* event, M_{sel} initiates data sending service by producing *SendData* event and moves to `SendingData`. The state machine waits for t_0 to receive the acknowledgment for the transmitted data by the server. If the acknowledgment does not arrive, it jumps to `ExecutingLocal`, whereas in the other case, the state machine transitions to `ExecutingRemote` and waits for the final response. If the response comes within t_4 secs, state machine jumps to `SyncWithSLO` and waits for the next cycle. However, if the response does not come within the deadline, an SLO violation is noted.

If the asynchronous process, p, produces *SwitchToLocal* or does not emit any signal within time interval t_5 then M_{sel} jumps to `ExecutingLocal` from `SelectingMode`. While transitioning to `ExecutingLocal`, the state machine generates an event, *ProcessDataLocal* to trigger local data processing service. If the data is not processed with in t_1 secs, *TimeOut*(t_1) forces the state machine to move to `SyncWithSLO` and SLO violation is noted again. On the other hand if t_1 deadline is not violated, state machine also moves to back `SyncWithSLO` and waits till the next cycle starts.

5. Experimental Validation

We now present the results of empirically evaluating URMILA's capabilities and validating the claims we made by answering the following questions:

- How effective is URMILA's execution time estimation on heterogeneous hardware? §5.3.1
- How effective is URMILA's connectivity and network latency estimation considering user mobility? §5.3.2
- How effective is URMILA in assuring SLOs? §5.3.3
- How much energy can URMILA save for mobile user?§5.3.3
- How does URMILA compare to other algorithms?§5.3.3

5.1. IoT Application Use Case

We assume the applications are containerized and can be deployed across edge and fog/cloud thereby eliminating the need to continuously re-deploy the application logic between the fog and edge devices. However, for platforms such as Android cannot yet run containers, a separate implementation for Android device and fog/cloud are used and it is just a matter of dynamically (de)activating the provisioned task on either the edge or fog device based on URMILA's resource management decisions.

For the experimental evaluation, we use the cognitive navigational assistance use case from Section 2.2. Since similar use cases reported in the literature are not available for research or use obsoleted technologies, and also to demonstrate the variety in the edge devices used, we implemented two versions of the same application. The first implementation uses an Android smartphone that inter-operates with a Sony SmartEyeGlass, which is used to capture video frames as the user moves in a region and provides audio feedback after processing the frame. The second version comprises a Python application running on Linux-based board devices such as MinnowBoard with a Web camera. The edge-based and fog-based image processing tasks implement MobileNet and Inception V3 real-time object detection algorithms from Tensorflow, respectively.

For our evaluations we assume that users of URMILA will move within a region, such as a university campus, with distributed WAPs or wireless hotspots owned by internet service providers some of which will have an associated MDC. We also assume an average speed of 5 kms/hour or 3.1 miles/hour for user mobility while accessing the service.[4] Note that URMILA is not restricted to this use case alone nor to the considered user mobility speeds. Empirical validations in other scenarios remain part of our future work.

5.2. Experimental Setup

We create two experimental setups to emulate realistic user mobility for our IoT application use case as follows:

[4] `https://goo.gl/cMxdtZ`

First Setup: We create an indoor experimental scenario with user mobility emulated over a small region and using our Android-based client. The Android client runs on a Motorola Moto G4 Play phone with a Qualcomm Snapdragon 410 processor, 2 GB of memory and Android OS version 6.0.1. The battery capacity is 2800 mAh. It is connected via bluetooth to Sony SmartEyeglass SED-E1 which acts as both the sensor for capturing frames and the actuator for providing the detected object as feedback. The device can be set to capture the video frames at variable frames per second (fps). We used a Raspberry Pi 2B running OpenWRT 15.05.1 as our WAP, which operates at a channel frequency of 2.4 GHz.

We set the application SLO to 0.5 second based on a previous study, which reported mean reaction times to sign targets to be 0.42-0.48 second in one experiment and 0.6-0.7 second in another [23]. Accordingly, we capture the frames at 2 fps, while the user walking at 5 kms/hour expects an update within 500 ms if the detected object changes.

Second Setup: We emulate a large area containing 18 WAPs, four of which have an associated MDC. We experiment with different source and destination scenarios and apply the latency estimation technique to estimate the signal strength at different segments of the entire route. We then use three OpenWRT-RaspberryPi WAPs to emulate the signal strengths over the route by varying the transmit power of the WAPs at the handover points, i.e., where the signal strength exceeds or drops below the threshold of -67 dBm. We achieve this by creating a mapping of the received signal strength on the client device at the current location and varying the transmit power of the WAP from 0 to 30 dBm.

For the client, we use our second implementation comprising Minnowboard Turbot, which has an Intel Atom E3845 processor with 2 GB memory. The device runs Ubuntu 16.04.3 64-bit operating system and is connected to a Creative VF0770 webcam and Panda Wireless PAU06 WiFi adapter on the available USB ports. In this case too, we capture the frames at 2 fps with a frame size of 224x224. To measure the energy consumption, we connect the Minnowboard power adapter to a Watts Up Pro power meter. We measure the energy consumption when our application is not running, which on average is 3.37 Watts. We then run our application and measure the power every second. By considering the power difference in both scenarios, we derive the energy consumption per period for a duration of 500 ms.

Application Task Platform: The Android device runs Tensorflow Light 1.7.1 for the MobileNet task. The Linux client runs the task in a Docker container. We use this model so that we can port the application across platforms and benefit from Docker's near native performance [24]. We use Ubuntu 16.04.3 containers with Keras 2.1.2 and Tensorflow 1.4.1.

Micro Data Center Configuration: For the deployment, we use heterogeneous hardware configurations shown in Table 2. The servers have different number of processors, cores and threads. Configurations F, G and H also support hyper-threads but we disabled them in our setting. We randomly select from a uniform distribution of the 16 servers specified in Table 2 and assign four of them to each MDC. In addition, for each server, the interference load and their profiles are selected randomly such that the servers have medium to high load without any resource over-commitment, which is typical of data centers [25]. Although the MDCs are connected to each other over LAN in our setup, to emulate WANs with multi-hop latencies, we used `www.speedtest.net` on intra-city servers for ping latencies and found 32.6 ms as the average latency. So, we added 32.6 ms ping latency with a 3 ms deviation between WAPs using the *netem* network emulator.

The Docker guest application has been assigned 2 GB memory and 4 CPU-pinned cores. For our experimentation, we use a server application that listens on TCP port for receiving the images and sending the response. Please note, our framework is independent of the communication mechanism as long as we have an accurate measure of network latency for the size of data transferred. Thus, we could also support UDP (unreliable) and HTTP (longer latency).

The size of a typical frame in our experiment is 30 KB. For the co-located workloads that cause performance interference, we use 6 different test applications from the Phoronix test suite (`www.phoronix-test-suite.com/`), which are either CPU, memory or disk intensive, and our target latency-sensitive applications, which involve Tensorflow inference algorithms.

5.3. Empirical Results

To obtain the response time, we need the edge-based task execution time, and the fog-based execution time plus network delay. In Equation (3), there are three main components, $t_{local}(u)$, $t_{remote}(u, s)$ and $t_{network}(u, s, p)$ and we need accurate estimates of all three at deployment time such that we could adhere to SLO requirements. $t_{local}(u)$ has negligible variations as long as the client device is running only the target application u which is a fair assumption for the mobile devices.

When the MinnowBoard Linux client device processes a 224x224 frame, the measured mean execution

Table 2: Server Architectures

Conf	sockets/cores/ threads/ GHz	L1/L2/L3 Cache(KB)	Mem Type/ MHz/GB	Count
A	1/4/2/2.8	32/256/8192	DDR3/1066/6	1
B	1/4/2/2.93	32/256/8192	DDR3/1333/16	2
C	1/4/2/3.40	32/256/8192	DDR3/1600/8	1
D	1/4/2/2.8	32/256/8192	DDR3/1333/6	1
E	2/6/1/2.1	64/512/5118	DDR3/1333/32	7
F	2/6/1/2.4	32/256/15360	DDR4/2400/64	1
G	2/8/1/2.1	32/256/20480	DDR4/2400/32	2
H	2/10/1/2.4	32/256/25600	DDR4/2400/64	1

times for MobileNet and Inception V3 are 434 ms and 698.6 ms, with standard deviations of 8.6 ms and 12.9 ms, respectively.

Since we have already measured the efficacy of NUMA-aware deployment in Figure 5, we employ NUMA-awareness in all the experimental scenarios.

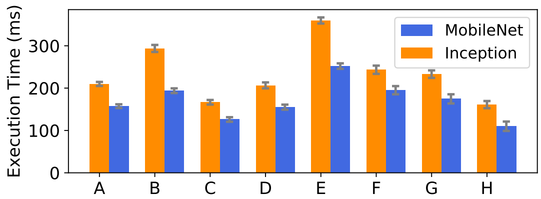
5.3.1. *Accuracy of Performance Estimation*

We report on the accuracy of the offline learned performance models. For $t_{remote}(u, s)$, in addition to hardware type w, we also consider the server load. We first measure $t_{isolation}(u, w)$ for each hardware type given in Table 2, and the results are shown in Figure 7a. We observe that the CPU speed, memory and cache bandwidth and the use of hyper-threads instead of physical cores play a significant role in the resulting performance. Thus, the use of a per-hardware configuration performance model is a key requirement met by URMILA. We also profile the performance interference using gradient tree boosting regression model with tools we developed in [7].

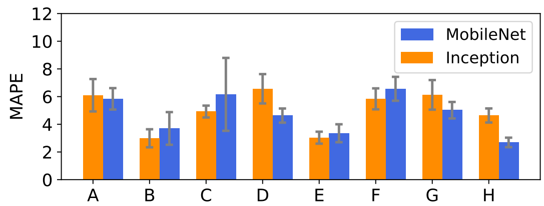
Figure 7b shows the estimation errors on different hardwares, which are well within 10% and hence can be used in our response time estimations by allowing for a corresponding margin of error.

5.3.2. *Accuracy of Latency Estimation*

We evaluate the accuracy of URMILA's network latency estimation module that calculates $t_{network}(u, s, p)$. From Equation (2), there are two main components to it: last-hop latency, $t_{u,ap(u,p)}$ and WAN latency, $t_{ap(u,p),ap(s)}$. $t_{ap(u,p),ap(s)}$ remains stable over a duration of time [26, 27] which is sufficient for URMILA scenarios and we emulate these as described in Section 5.2. Thus, we are left with $t_{u,ap(u,p)}$. As the received signal strength is a key factor for last hop latency, we determine γ for Equation (14) for a typical access point described in Section 5.2 for the indoor environment of our lab. We used the Android device to measure signal strength and network latency for the used data transfer size. Figure 8a shows



(a) Execution Time in Isolation

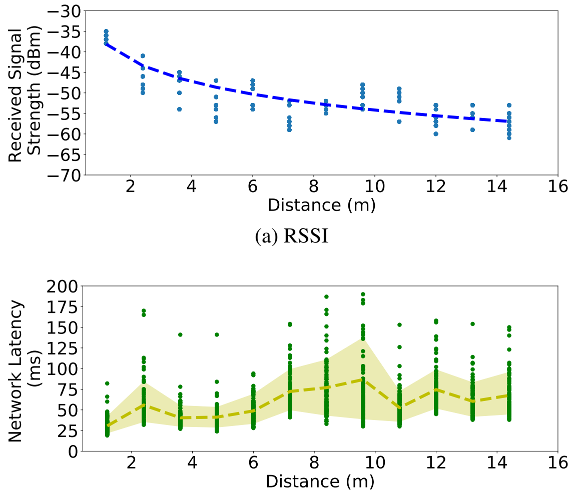


(b) Mean Absolute Percentage Error

Figure 7: Performance Estimation Model Evaluations

the results where we found γ to be 1.74, inline with the expected indoor value of 1.6-1.8 as described in Section 4.3. Figure 8b, affirms our assertion that network latency remains near constant within a fixed range of received signal strength.

Next, we measure network latency for five different routes on our selected campus area with 18 WAPs. We chose $\gamma = 2$ for outdoors [17] and generated varied signal strengths for the entire path on five routes. Using these values, we setup the WAPs such that the client device experiences WAP handovers and regions with no connectivity. Figure 9 shows the results for the five routes (R1–R5). The shaded areas show the regions with no network connectivity and regions with different colors show connectivity to different WAPs. The green line is the signal strength and the black line is the mean latency. There are gaps in latency values, which indicate that the client device is performing handover to the access point. We observe from these plots that even though the mean latency values are low when connected to the wireless network, there are large latency deviations. For example, on route R1 at $t = 400$s, the mean



(a) RSSI

(b) Network Latency

Figure 8: Signal Strength and Network Latency Variations with Distance

latency is 52ms but the 99th percentile latency is 384ms. Hence, for ensuring SLOs, we need to use the required SLO percentile values from our database of network latencies on the user's route as described in Algorithm 1.

5.3.3. *Efficacy of URMILA's Fog Server Selection*

We evaluate how effective is URMILA's server selection technique in ensuring that SLOs are met. We evaluate the system for the five routes described above and set four of the 18 available access points as MDCs and assign servers as described in Section 5.2. We compare URMILA against different mechanisms. One approach is when we perform everything locally (*Local*), and another approach is the maximum network coverage (*Max Coverage*) algorithm, where the server is selected based on the network connectivity.

For this set of experiments, we keep the deployment (Equation (4)) and transfer (Equation (5)) costs constant in our Algorithm 1 for all the scenarios. We also set the required SLO at 95^{th} percentile of the desired response time of 500ms (2 fps). We then optimize for energy consumption (Equation (6)) while meeting the constraints (Equations (8)-(13)).

Figure 10a reveals that if we run higher accuracy Inception as the target application, the *Local* mode always misses the deadline of 500ms, however, the lower accuracy MobileNet always meets the deadline (Figure 10b). Nevertheless, from Figure 11 we observe that while executing higher accuracy Inception V3 algorithm, URMILA consumed 39.61% less energy compared to *Local* mode on an average . Figure 10d shows that URMILA meets the SLO 95% of the time for all routes while consuming 9.7% less energy in comparison to *Max Coverage* (Figure 10c).

The *Max Coverage* algorithm performed worse than URMILA for energy consumption and on 4 out of 5 routes for response time consumes 9.7%. For these experiments *Least loaded* performs at par with URMILA. Please note as URMILA considers both the server load and and network coverage, it will perform at least at par to the other two techniques for assuring SLOs.

We now demonstrate the scenario when URMILA performs better that *Least loaded*. In our current experimental setup, we considered there is similar latencies between the access points $t_{ap(u,\ell),ap_i}$ and for the last hop, $t_{u,ap(u,\ell)}$ channel utilization and connected users are less. However, this is not usually the case. Thus, we introduce use a latency value of 100.0ms with 10% deviation for some of the access points. In real deployments, URMILA will be aware of this latency by WAP to WAP measurements. Thus, as depicted in Figure 12, for *Least Loaded*, SLOs will be violated even for best performing server due to the ignorance about the network communication delay. However, URMILA's robust runtime component is aware of the deployment plan and performs execution locally for the WAPs that cannot meet the constraints.

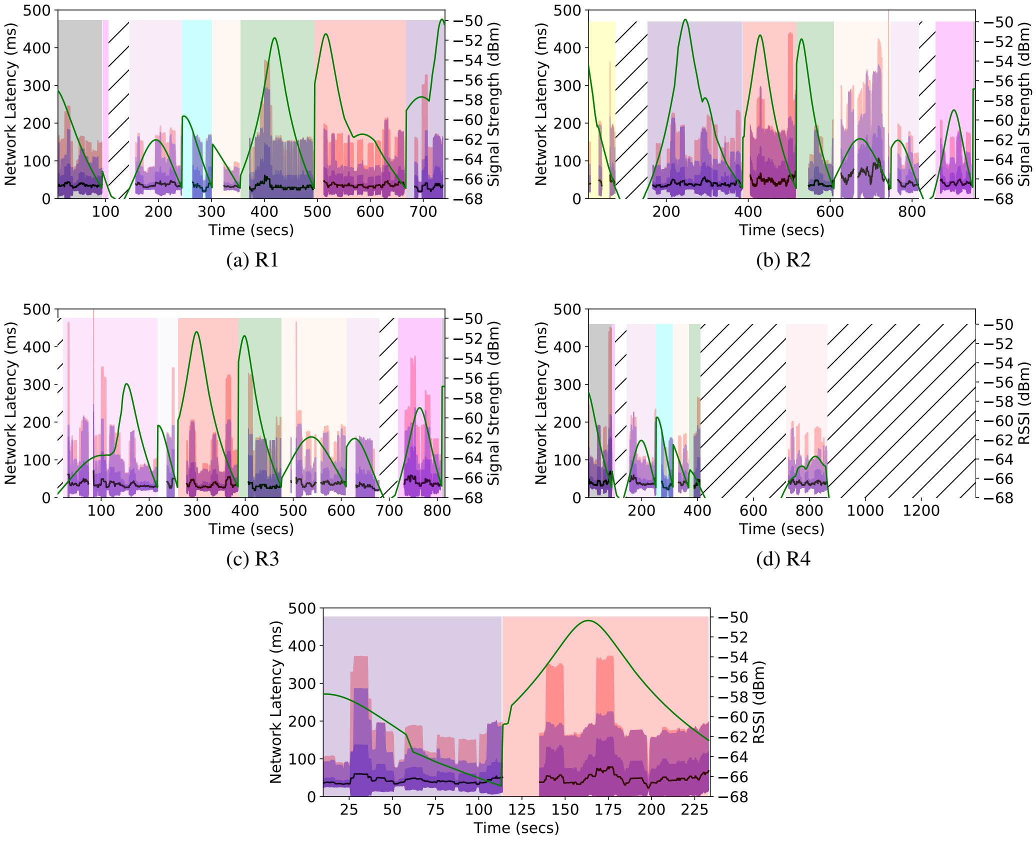
In the above experiments, we considered that there is sufficient gap between when the user requests the service and when she actually needs it. However, this may not be true and we need to consider the transfer and initialization costs of Equation (5). We setup Docker private registry and shaped the network bandwidth such that we could do the measurements for image overlays being transferred of different sizes. Table 3 depicts the same.

Table 3: Transfer and Initialization Cost Measurements

Image	Size (MB)	Duration at 10 Mbps	Duration at 1 Mbps
Cached	-	13.2s	13.46s
Overlay 1	111	31.6s	127.08s
Overlay 2	440	50.26s	261.87s

6. Related Work

Since URMILA considers the three dimensions of performance interference issues, mobility-aware resource management and exploiting edge/fog holistically, we provide a sampling of the prior work in these areas and compare the URMILA solution with these efforts. An earlier, shorter version of the URMILA work



(a) R1
(b) R2
(c) R3
(d) R4
(e) R5

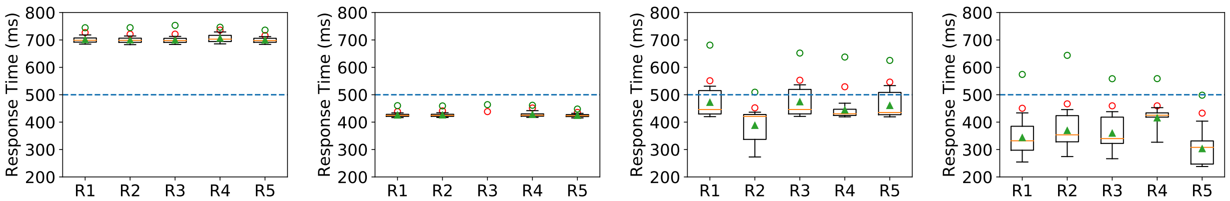
Figure 9: Observed Mean, Std Dev, 95th and 99th Percentile Network Latencies and Received Signal Strengths on Emulated Routes

appears in [9]. This paper significantly improves upon the earlier version by providing an optimization problem formulation, more details on the latency estimation and effects of corepinning, and detailed steps during run-time. To the best of our knowledge, we have not found any prior efforts that consider all these three dimensions simulataneously.

6.1. Performance Interference-aware Resource Optimization

There have been a number of prior efforts that account for performance interference during server selection to host cloud jobs. Bubble-Flux [4] is a dynamic interference measurement framework that performs online QoS management while maximizing server utilization and uses a dynamic memory bubble for profiling by pausing other co-located applications. Freeze'nSense [28] is another approach that performs a short duration freezing of interfering co-located tasks. The advantage of an online solution is that an *a priori* knowledge of the target application is not required and it does not need additional hardware resources for benchmarking. Although in these works, *a priori* knowledge of the target application is not required nor extra benchmarking efforts, pausing (even for short duration) of co-located applications is not desirable and in several cases not even possible as these applications will have their own SLOs to be met.

DeepDive [29] is a benchmarking based effort that identifies the performance interference profile by cloning the target VM and benchmarking it when QoS violations are encountered. However, this is too expensive an operation to be employed at run-time. Paragon [2] is a heterogeneity- and interference-aware data center scheduler the applies analytical techniques to reduce the benchmarking workload. URMILA falls in this category of work, nevertheless, it goes a step further and also considers scheduler-specific metrics which play a significant role in accurate performance estimation on multi-tenant platforms.



(a) Local Inception (b) Local Mobilenet (c) Max Coverage Inception (d) URMILA Inception

Figure 10: Response Time for Different Techniques on the Routes. ○ and ○ depict the 95th and 99th percentile, respectively

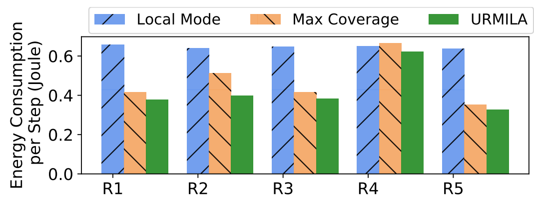


Figure 11: Energy Consumption Comparison

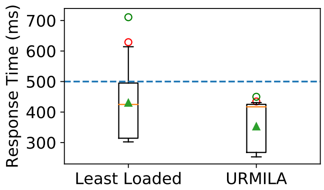


Figure 12: Response Time Comparison for Route R5 when one of the WAP is Experiencing Larger Latency

6.2. Mobility-aware Resource Management

MOBaaS [30] is a mobile and bandwidth prediction service based on dynamic Bayesian networks. Sousa et al. [6] utilize MOBaaS to enhance the follow-me cloud (FMC) model, where they first perform mobility and bandwidth prediction with MoBaaS and then apply a multiple attribute decision algorithm to place services. However, this approach needs a history of mobility patterns by monitoring the users. URMILA currently uses a deterministic path for the user, which provides a more accurate and efficient solution. However, future work will explore probabilistic routes taken by the mobile user.

MuSIC defines applications as location-time workflows, and optimizes their QoS expressed as the power of the mobile device, network delay and price [31]. Like MuSIC, URMILA aims to minimize energy consumption of edge devices, communication costs, and cost of operating fog resources. Unlike MuSIC, which evaluates its ideas via simulations, URMILA has been evaluated empirically. In addition, MuSIC assumes certain variations in network patterns without applying any prediction/estimation methodology, while URMILA provides concrete capabilities to predict/estimate network behavior.

Additional prior work includes [32], which considers different classes of mobile applications and apply three scheduling strategies on fog resources. Likewise, Wang et al. [33] account for user mobility and provide both offline and online solutions for deploying service instances considering a look-ahead time-window. Both these approaches do not consider edge resources for optimization as we do in URMILA. Similarly, ME-VoLTE [34] is an approach to offload video encoding from mobile devices to cloud for reducing energy consumption. However, the approach does not consider latency issues when offloading.

6.3. Resource Management involving Fog/Edge Resources:

Cloudlet [1] is a miniature data center closer to the user, possibly just one wireless hop away, that is meant to overcome the latency issues faced by edge-based applications that must use centralized cloud resources that are many network hops away. This vision was refined into a three tier architecture [8] comprising the edge, fog and cloud tiers. This is the model used by URMILA.

CloudPath [35] expands on the cloud-fog-edge architecture [8] by proposing the notion of *path computing* comprising *n* tiers between the edge and the cloud, where applications can be dynamically hosted to meet their processing and storage requirements. CloudPath requires applications to be stateless and made up of short-lived functions – similar to the notion of *function-as-a-service*, which is realized by serverless computing solutions with state in externalized databases. We believe that the research foci of CloudPath and URMILA are orthogonal; the CloudPath platform and its path computing paradigm can potentially be used by URMILA to host its services and by incorporating our optimization algorithm in CloudPath's platform.

The LAVEA project [36] comes close to our vision of URMILA yet their goals are complementary. LAVEA supports a video analytics framework that executes in the fog/edge hierarchy similar to URMILA. They use a slightly different terminology referring to the edge devices as mobile devices, and fog devices as edge devices. "Edge-first" (i.e., execute on the fog resources) is the main philosophy for LAVEA. Like CloudPath, LAVEA also leverages serverless computing thereby requiring stateless applications. LAVEA focuses on scheduling and prioritizing tasks on the fog resources when multiple, independent client jobs get offloaded to fog nodes. It also supports coordination among fog nodes. While URMILA can certainly benefit from LAVEA's fog node scheduling algorithms, it focuses on ensuring SLOs of individual services and makes every effort to maintain high availability of the service by executing it either on the edge or the fog node, and moreover, also allows mobility of users.

Precog [37] is another edge-based image recognition system. Like URMILA they also recognize the need to conserve battery resources on edge devices and hence can perform selective image recognition on the edge devices. To speed up execution on fog nodes, they support the notion of the so called *recognition cache,* which prefetch only parts of the trained models that are used to recognize images. Unlike Precog, URMILA performs these tasks by maintaining two different versions of the service: one that can execute on the edge and one on the fog, and dynamically switches between them to meet the SLOs.

Our prior work called INDICES [7] is an effort that exploits the cloud-fog tiers. INDICES decides the best cloudlet (i.e., fog resource) and the server within that cloudlet to migrate a service from the centralized cloud so that SLOs are met. INDICES does not handle user mobility and its focus is only on selecting an initial server on a fog resource to migrate to. It does not deal with executing tasks on the edge device. Thus, URMILA's goals are to benefit from INDICES' capabilities by exploiting its initial server selection in the fog layer and extend it by intelligently adapting between fog and edge resources while supporting user mobility.

7. Conclusion

Although fog/edge computing have enabled low latency edge-centric applications by eliminating the need to reach the centralized cloud, solving the performance interference problem for fog resources is even harder than traditional cloud data centers. User mobility amplifies the problem further since choosing the right fog device becomes critical. Executing the service at all times exclusively on the edge devices or fog resources is not an alternative either. This paper presented URMILA to holistically address these issues by adaptively using edge and fog resources to make trade-offs while satisfying SLOs for mobility-aware IoT applications.

7.1. Discussion and Broader Impact

URMILA has broader applicability beyond cognitive assistance application that is evaluated in this work. For instance, URMILA can be used in cloud gaming (such as Pokemon GO), 3D modeling, graphics rendering, etc. We could apply URMILA for energy efficient route selection and navigation. For that, we can easily modify Algorithm 1 to find the most energy efficient route.

By no means does URMILA address all the challenges in this realm and our future work will involve: (a) considering probabilistic routes taken by the user; (b) evaluating URMILA in other applications, e.g., smart transportation where the speed is higher and distances covered are larger so choosing only one fog server at initialization may not be feasible; (c) leveraging the benefits stemming from upcoming 5G networks; and (d) showcasing URMILA's strengths in the context of multiple competing IoT applications.

The software and experimental setup of URMILA is available in open source at `github.com/doc-vu`.

7.2. Opportunities for Future Work

The following form the dimensions of our future work.

Last Hop latency: For un-profiled routes, we only considered received signal strength for wireless network latency estimation. However, channel utilization and connected users play a significant role in latency variations. To overcome this potentially less accurate latency estimation, we can collect these metrics from WAPs, but this will require access to their data. Other option is to use a predictive approach based on data collected for other profiled routes.

Speed of mobility and route determination: For the user mobility, we considered constant speed mobility and deterministic routes, however, in general the user can deviate from the ideal route and have a varying velocity. This may render the initial deployment plan suboptimal. We account for this in our server allocation, but, the runtime algorithm can further be improved to intelligently adjust the route plan based on current dynamics and probabilistic routes.

Overhead: URMILA incurs cost for both the client device and the service provider due to metric collection on each server. The overhead of INDICES monitoring agents [7] is $\approx 1\%$. We also need to maintain a database of performance metrics at each MDC and the *gm* needs to perform learning. In addition, the cost of profiling each new application may not be insignificant depending on the lifespan of the application, However, this is a one time cost and is required for overcoming performance interference. On the client device, we made a conscious effort to not to use GPS coordinates while the user is mobile. This is because GPS has significant energy overhead and we did not want our application to be limited to navigational applications. In addition, turning on wireless and handovers are expensive. However, most mobile devices have their wireless service turned on these days, so we do not consider it as additional cost.

Serverless Computing: Since we target containerized stateless applications, we could potentially make our solution apt for serverless computing, wherein the same containers are shared by multiple users and the application scale as the workload varies, and are highly available.

Future Direction: Apart from what we discussed, our solution can be enhanced by controlling frame rates based on the user needs and location. We considered monolithic applications, we could allocate services with multiple components that are deployed across the spectrum optimally. In future, we could address concerns related to trust, privacy, billing, fault tolerance and workload variations.

Acknowledgments

This work was supported in part by NSF US Ignite CNS 1531079, AFOSR DDDAS FA9550-18-1-0126 and AFRL/Lockheed Martin's StreamlinedML program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these funding agencies.

References

[1] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, P. Pillai, Cloudlets: At the Leading Edge of Mobile-Cloud Convergence, in: Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on, IEEE, 2014, pp. 1–9.

[2] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware Scheduling for Heterogeneous Datacenters, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 77–88.

[3] W. Kuang, L. E. Brown, Z. Wang, Modeling Cross-Architecture Co-Tenancy Performance Interference, in: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2015, pp. 231–240.

[4] H. Yang, A. Breslow, J. Mars, L. Tang, Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers, in: ACM SIGARCH Computer Architecture News, Vol. 41, ACM, 2013, pp. 607–618.

[5] S. Shekhar, H. A. Aziz, A. Bhattacharjee, A. Gokhale, X. Koutsoukos, Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications, in: IEEE International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2018, pp. 82–89.

[6] B. Sousa, Z. Zhao, M. Karimzadeh, D. Palma, V. Fonseca, P. Simoes, T. Braun, H. Van Den Berg, A. Pras, L. Cordeiro, Enabling a Mobility Prediction-Aware Follow-Me Cloud Model, in: Local Computer Networks (LCN), 2016 IEEE 41st Conference on, IEEE, 2016, pp. 486–494.

[7] S. Shekhar, A. Chhokra, A. Bhattacharjee, G. Aupy, A. Gokhale, INDICES: Exploiting Edge Resources for Performance-Aware Cloud-Hosted Services, in: IEEE 1st International Conference on Fog and Edge Computing (ICFEC), Madrid, Spain, 2017, pp. 75–80. doi:10.1109/ICFEC.2017.16.

[8] M. Satyanarayanan, Edge Computing: a New Disruptive Force, Keynote Talk at the 3rd ACM/IEEE International Conference on Internet of Things Design and Implementation (IoTDI) (Apr. 2018).

[9] S. Shekhar, A. D. Chhokra, H. Sun, A. Gokhale, A. Dubey, X. Koutsoukos, URMILA: Dynamically Trading-off Fog and Edge Resources for Performance and Mobility-Aware IoT Services, in: IEEE Symposium on Real-time Computing (ISORC 2019), Valencia, Spain, 2019, pp. 118–125. doi:10.1109/ISORC.2019.00033.

[10] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, A close examination of performance and power characteristics of 4g lte networks, in: Proceedings of the 10th international conference on Mobile systems, applications, and services, ACM, 2012, pp. 225–238.

[11] C. Gray, R. Ayre, K. Hinton, R. S. Tucker, Power Consumption of IoT Access Network Technologies, in: Communication Workshop (ICCW), 2015 IEEE International Conference on, IEEE, 2015, pp. 2818–2823.

[12] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, H.-I. Yang, The case for cyber foraging, in: Proceedings of the 10th workshop on ACM SIGOPS European workshop, ACM, 2002, pp. 87–92.

[13] J. Mars, L. Tang, R. Hundt, K. Skadron, M. L. Soffa, Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations, in: 44th annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 248–259.

[14] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, Cpi2: Cpu performance isolation for shared compute clusters, in: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13, ACM, New York, NY, USA, 2013, pp. 379–391.

[15] C. Xu, X. Chen, R. P. Dick, Z. M. Mao, Cache contention and application performance prediction for multi-core systems, in: Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 76–86.

[16] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, T. Moscibroda, Characterizing and Improving WiFi Latency in Large-Scale Operational Networks, in: 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16, ACM, New York, NY, USA, 2016, pp. 347–360.

[17] T. S. Rappaport, et al., Wireless Communications: Principles and Practice, Vol. 2, prentice hall PTR New Jersey, 1996.

[18] S. Shekhar, M. Walker, H. Abdelaziz, F. Caglar, A. Gokhale, X. Koutsoukos, A Simulation-as-a-Service Cloud Middleware, Journal of the Annals of Telecommunications 74 (3-4) (2016) 93–108. doi:10.1007/s12243-015-0475-6.

[19] J. Rao, K. Wang, X. Zhou, C.-Z. Xu, Optimizing virtual machine scheduling in numa multicore systems, in: High Perfor-

mance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, IEEE, 2013, pp. 306–317.

[20] Intel resource director technology, last accessed: 11.08.2019. URL `https://github.com/intel/intel-cmt-cat/wiki`

[21] Y. Barve, S. Shekhar, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, A. Gokhale, FECBench: A Lightweight Interference-aware Approach for Application Performance Modeling, in: IEEE International Conference on Cloud Engineering (IC2E), Prague, Czech Republic, 2019, pp. 211–221.

[22] J. Elith, J. R. Leathwick, T. Hastie, A working guide to boosted regression trees, Journal of Animal Ecology 77 (4) (2008) 802–813.

[23] K. G. Hooper, H. W. McGee, Driver Perception-Reaction Time: Are Revisions to Current Specification Values in Order?, Tech. rep. (1983).

[24] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On, IEEE, 2015, pp. 171–172.

[25] L. A. Barroso, J. Clidaras, U. Hölzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines, Synthesis lectures on computer architecture 8 (3) (2013) 1–154.

[26] S. K. Barker, P. Shenoy, Empirical evaluation of latency-sensitive application performance in the cloud, in: Proceedings of the first annual ACM SIGMM conference on Multimedia systems, ACM, 2010, pp. 35–46.

[27] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, A. Pescapè, Broadband Internet Performance: A View from the Gateway, in: ACM SIGCOMM computer communication review, Vol. 41, ACM, 2011, pp. 134–145.

[28] A. Kandalintsev, D. Kliazovich, R. Lo Cigno, Freeze'nsense: estimation of performance isolation in cloud environments, Software: Practice and Experience.

[29] D. Novaković, N. Vasić, S. Novaković, D. Kostić, R. Bianchini, DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments, in: USENIX Conference on Annual Technical Conference, USENIX ATC'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 219–230. URL `http://dl.acm.org/citation.cfm?id=2535461.2535489`

[30] M. Karimzadeh, Z. Zhao, L. Hendriks, R. d. O. Schmidt, S. la Fleur, H. van den Berg, A. Pras, T. Braun, M. J. Corici, Mobility and Bandwidth Prediction as a Service in Virtualized LTE Systems, in: Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on, IEEE, 2015, pp. 132–138.

[31] M. R. Rahimi, N. Venkatasubramanian, A. V. Vasilakos, MuSIC: Mobility-aware Optimal Service Allocation in Mobile Cloud Computing, in: Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, IEEE, 2013, pp. 75–82.

[32] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, M. Parashar, Mobility-aware application scheduling in fog computing, IEEE Cloud Computing 4 (2) (2017) 26–35.

[33] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, K. K. Leung, Dynamic service placement for mobile micro-clouds with predicted future costs, IEEE Transactions on Parallel and Distributed Systems 28 (4) (2017) 1002–1016.

[34] M. T. Beck, S. Feld, A. Fichtner, C. Linnhoff-Popien, T. Schimper, Me-volte: Network functions for energy-efficient video transcoding at the mobile edge, in: Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on, IEEE, 2015, pp. 38–44.

[35] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, E. de Lara, CloudPath: A Multi-Tier Cloud Computing Framework, in: Second ACM/IEEE Symposium on Edge Computing, ACM, 2017, p. 20.

[36] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, Q. Li, Lavea: Latency-aware video analytics on edge computing platform, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, ACM, 2017, p. 15.

[37] U. Drolia, K. Guo, P. Narasimhan, Precog: prefetching for image recognition applications at the edge, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, ACM, 2017, p. 17.