2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)

URMILA: A Performance and Mobility-Aware Fog/Edge Resource Management Middleware

Shashank Shekhar*‡, Ajay Chhokra†, Hongyang Sun†, Aniruddha Gokhale†,
Abhishek Dubey† and Xenofon Koutsoukos†
*Siemens Corporate Technology, Princeton, NJ 08540 and †Vanderbilt University, Nashville, TN 37235
Email: *shashankshekhar@siemens.com and †{ajay.d.chhokra,hongyang.sun,a.gokhale,
abhishek.dubey,xenofon.koutsoukos}@vanderbilt.edu

***Abstract*—Fog/Edge computing is increasingly used to support a wide range of latency-sensitive Internet of Things (IoT) applications due to its elastic computing capabilities that are offered closer to the users. Despite this promise, IoT applications with user mobility face many challenges since offloading the application functionality from the edge to the fog may not always be feasible due to the intermittent connectivity to the fog, and could require application migration among fog nodes due to user mobility. Likewise, executing the applications exclusively on the edge may not be feasible due to resource constraints and battery drain. To address these challenges, this paper describes URMILA, a resource management middleware that makes effective trade-offs between using fog and edge resources while ensuring that the latency requirements of the IoT applications are met. We evaluate URMILA in the context of a real-world use case on an emulated but realistic IoT testbed.**

***Index Terms*—Fog Computing, Edge Computing, Ubiquitous Computing, User Mobility, Latency-sensitive, IoT Services, Resource Management, Energy Consumption.**

I. 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 essentially a miniaturized data center and hence supports multi-tenancy and elasticity, however, at a limited scale and 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 the information requires sufficient computational capabilities. Thus, exclusively using edge resources for these computations may not always be feasible because either 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 intensive and how long such computations can be carried. In contrast, exclusive use of cyberforaging, i.e., offloading the computations to the fog layer is not a solution because offloading of data incurs communication 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 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 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 as the user moves 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 application. 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[1] and to ensure that SLO violations are minimized for latency-sensitive IoT applications, particularly those that are utilized in mobile environments.

The rest of this paper is organized as follows: Section II discusses the application and the system models; Section III explains the URMILA solution in detail; Section IV provides empirical validation of our work; Section V describes related work in comparison to URMILA; and finally Section VI provides concluding remarks.

‡ Work performed by the first author during doctoral studies at Vanderbilt University.

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

978-1-7281-0151-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ISORC.2019.00033

II. System Model and Assumptions

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

A. 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 [9].

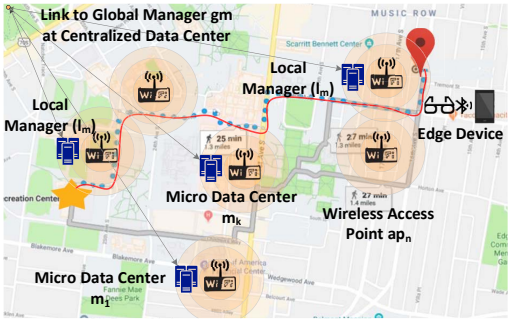


Fig. 1: System infrastructure model

B. Application Model

We present our IoT application model via a use case comprising 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. 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. 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. Application execution is assumed to be made up of a sequence of individual, repeated tasks of approximately equal length (e.g., frame capture, processing and feedback steps of our use case).[2] 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 that does not 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.

C. 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. 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.

III. URMILA: Design and Implementation

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

A. Overview of URMILA's 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. 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 traversing the WAN links. 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

[2]Tasks are assumed to be periodic, but our work is not limited to periodic tasks.

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 it.

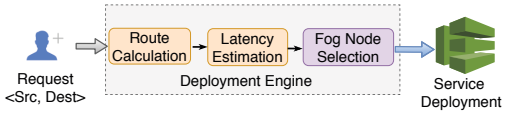


Fig. 2: Components for fog server selection

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.

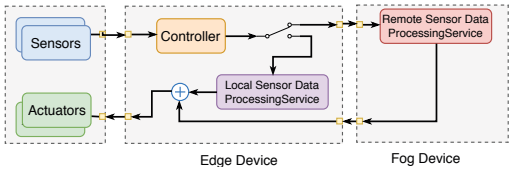


Fig. 3: URMILA architecture for decision making

B. Latency Estimation: Rationale and Approach

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: the last-hop latency to a WAP along the route, the WAN latencies to reach the fog server by traversing the WAN links, and the task execution time on the fog server (See Section III-C).

Estimating Last-hop Latency: The last hop latency itself is affected primarily by *channel utilization*, *number of active users* and *received signal strength* [10]. 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.

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 [10], which we use as the threshold for URMILA. We also use the well-known methods for determining the signal strength based on received power and distance from an access point [11]. 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: The WAN latency between two WAPs 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.

C. Selecting the Most Suitable Fog Server

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 to-be selected fog server, which will depend on the instantaneous co-located workloads on that server and the incurred performance interference. For this, we leverage our INDICES performance metric collection and interference modeling framework [7].

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

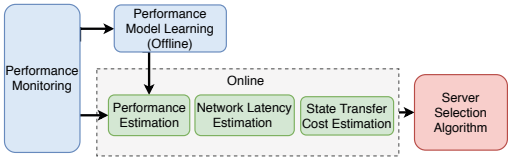


Fig. 4: Fog server selection process

Offline Performance Model Learning: Since URMILA uses a data-driven approach in its run-time decision making, it uses offline training to develop a performance model for each latency-sensitive application task that gets executed on the

fog server. The model depends on two factors: (1) the server architecture and configuration, which is a leading cause of performance variability [2], and (2) the application's sensitivity in the presence of co-located applications and its pressure on those applications [3] since no perfect isolation in shared multi-tenant environments is feasible due to non-partitionable resources such as shared last-level cache, interconnect network, disk and memory bandwidth [4].

To obtain a performance interference model, we first benchmark the execution time $t_{isolation}(u, w)$ of each latency-sensitive application u on a specific hardware type w in isolation. Then, we execute the application with different co-located workload patterns and learn its impact g_u on the system-level and obtain micro-architectural metrics as follows:

$$X_w^{new} = g_u(X_w^{cur}) \tag{1}$$

where X_w^{cur} and X_w^{new} denote the vectors of the selected metrics before and after running application u on hardware w, respectively. Lastly, we learn the performance deterioration f_u (compared to isolated performance) for application u under the new metric vector X_w^{new} on hardware w to predict its execution time on the fog server under the same conditions:

$$t_{remote}(u, w) = t_{isolation}(u, w) \cdot f_u(X_w^{new}) \tag{2}$$

We apply supervised machine learning based on Gradient Tree Boosting curve fitting to learn both functions g_u and f_u. This stage also involves feature selection, correlation analysis and regression technique selection. Note that Equations (1) and (2) can be applied together to model both sensitivity and pressure for application deployment on each server in order to accurately estimate remote execution times.

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.

Online Server Selection: The online stage performs server selection for an application as follows. First, when a user initiates a session, URMILA's global manager initiates the fog server selection process. For this, URMILA uses the route information and queries the local manager at each MDC on the user's route. The goal is to determine the expected execution time of the application task on each fog server in that 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. Next, the global manager combines this queried information with the latency estimates to determine which execution mode (local or remote) should the client-side use at each period/epoch of the application that meets all SLO constraints.

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. This execution plan will be used for cost estimation by the global manager and is subject to dynamic adjustment at run-time (See Section III-D).

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 ← ∞, s* ← ∅, and y*[p] ← 0 ∀p;
3      for each server s do
4          X^cur ← GetCurrentSystemMetrics(s);
5          X^new ← g_u(X^cur);
6          V ← GetListOfExistingApplications(s);
7          for each application v ∈ V do
8              t_process ← GetPreProcessingTime(v);
9              t_isolation ← GetIsolatedExecTime(v, s);
10             t_remote ← t_isolation · f_v(X^new);
11             t^SLO_network ← GetPercentileLatency(v, s);
12             if t_process + t_remote + t^SLO_network > φ(v) then
13                 skip s;
14             end
15         end
16         Initialize y[p] ← 0 ∀p;    // execute locally by default;
17         t_process ← GetPreProcessingTime(u);
18         t_isolation ← GetIsolatedExecTime(u, s);
19         t_remote ← t_isolation · f_u(X^new);
20         T_deploy ← GetDeploymentCost(u, s);
21         T_transfer ← GetStateTransferCost(u, s);
22         for each period p in the route do
23             t^SLO_network(p) ← GetPercentileLatency(u, s, p);
24             if t_process + t_remote + t^SLO_network(p) ≤ φ(u) then
25                 y[p] ← 1;        // execute this period remotely;
26             end
27         end
28         T_user ← ComputeUserCost(y);
29         cost ← α · T_deploy + β · T_transfer + κ · T_user;
30         if cost ≤ cost_min then
31             cost_min ← cost;
32             s* ← s and y* ← y;
33         end
34     end
35 end
```

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 considers the estimated network SLO percentile latency $t_{network}^{SLO}$ (e.g., 90$^{\text{th}}$, 95$^{\text{th}}$, 99$^{\text{th}}$) 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).

D. 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. 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. The mode selection strategy is described in more details in [12].

IV. Experimental Validation

We now present the results of empirically evaluating URMILA's capabilities and validating the claims we made.

A. IoT Application Use Case

For the experimental evaluation, we use the cognitive navigational assistance use case from Section II-B. 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.[3] 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.

[3]https://goo.gl/cMxdtZ

B. Experimental Setup

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

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 [13]. 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. 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 I. 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 I 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 [14]. 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.

TABLE I: Server Architectures

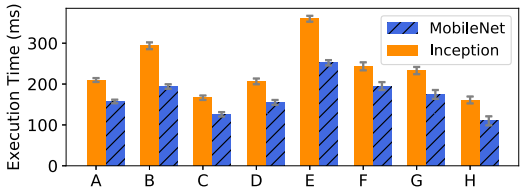
Conf	sockets/cores/ threads/ GHz	L1/L2/L3 Cache(KB)	Mem Type/ MHz/GB	Count
A	1/4/2/2.8	32/256/8192	DDR3/1066/6	1
B	1/4/2/2.93	32/256/8192	DDR3/1333/16	2
C	1/4/2/3.40	32/256/8192	DDR3/1600/8	1
D	1/4/2/2.8	32/256/8192	DDR3/1333/6	1
E	2/6/1/2.1	64/512/5118	DDR3/1333/32	7
F	2/6/1/2.4	32/256/15360	DDR4/2400/64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

The Docker guest application has been assigned 2 GB memory and 4 CPU-pinned cores. 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.

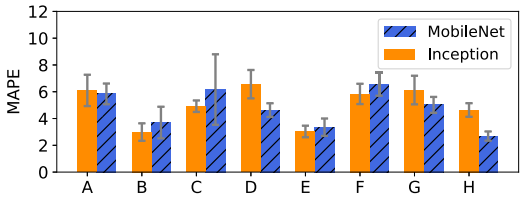
C. Empirical Results

To obtain the response time, we need the edge-based task execution time, and the fog-based execution time plus network delay. When the MinnowBoard Linux client device processes a 224x224 frame, the measured mean execution 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.

*1) **Accuracy of Performance Estimation**:* We report on the accuracy of the offline learned performance models. We first measure $t_{isolation}(u, w)$ for each hardware type given in Table I, and the results are shown in Figure 5a. 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 5b 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.



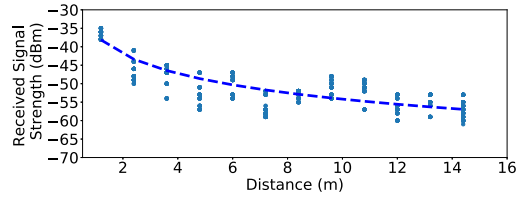
(a) Execution time in isolation



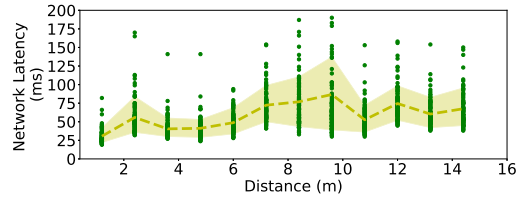
(b) Mean absolute percentage error

Fig. 5: Performance estimation model evaluations

*2) **Accuracy of Latency Estimation**:* We evaluate the accuracy of URMILA's network latency estimation module. From Section III-B, the total latency includes last hop, WAN latencies interconnecting WAPs and fog-based task execution. The WAN latencies tend to remain relatively stable over a long duration of time [15], which is sufficient for the URMILA scenarios and we emulate these as described in Section IV-B. Since the received signal strength is a key factor for last hop latency, we determine the path loss exponent γ [11] for a typical access point in our experimental setup also described in Section IV-B. We use the Android device to measure signal strength and network latency for the used data transfer size. Figure 6a shows the results where we found γ to be 1.74, which is inline with the expected indoor value of 1.6-1.8. Figure 6b affirms our assertion that network latency remains near constant within a fixed range of received signal strength.



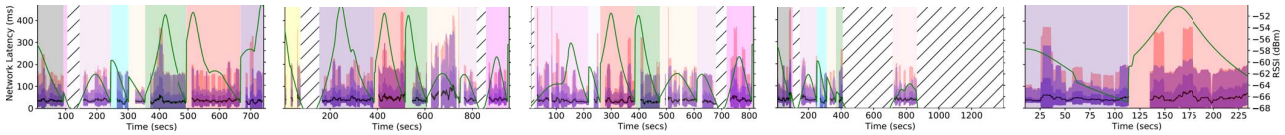
(a) RSSI



(b) Network latency

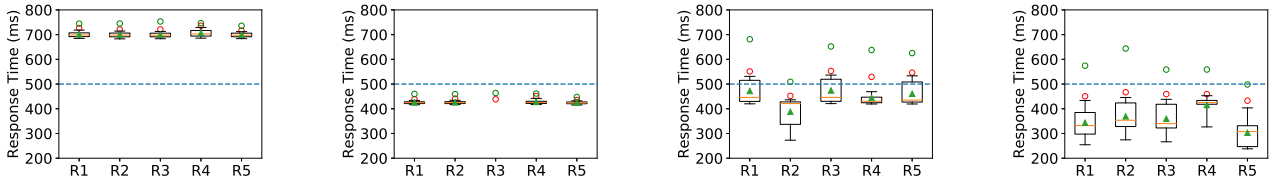
Fig. 6: Signal strength and network latency variations with distance

Next, we measure network latency for five different routes on our selected campus area with 18 WAPs. We chose γ = 2



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

Fig. 7: Observed mean, std dev, 95th and 99th percentile network latencies and received signal strengths on emulated routes



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

Fig. 8: Response time for different techniques on the routes. ○ and ○ depict the 95th and 99th percentile, respectively

for outdoors [11] 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 7 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 latency is 52 ms but the 99th percentile latency is 384 ms. 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.

*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 IV-B. 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. We keep the deployment and transfer costs in Algorithm 1 constant for all the scenarios. We also set the required SLO at 95th percentile of the desired response time of 500 ms (2 fps). We then optimize for energy consumption while meeting all the constraints. Figure 8a 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 8b). Figure 8d 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 8c).

V. Related Work

Since URMILA considers the three dimensions of performance interference issues, mobility-aware resource management and exploiting edge/fog holistically, we provide a brief sampling of the prior work in these areas.

Performance Interference-aware Resource Optimization: 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. Although in this work, *a priori* knowledge of the target application is not required nor extra benchmarking efforts, pausing of co-located applications is not desirable and in several cases not even possible. DeepDive [16] 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. 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.

Mobility-aware Resource Management: MOBaaS [17] 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 uses a deterministic path for the user, which provides a more accurate and efficient solution.

MuSIC defines applications as location-time workflows, and optimizes their QoS expressed as the power of the mobile device, network delay and price [18]. 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.

Resource Management involving Fog/Edge Resources: CloudPath [19] expands on the cloud-fog-edge architecture [8] by proposing the notion of *path computing* comprising n tiers between the edge and the cloud. 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.

Our prior work called INDICES [7] is an effort that exploits the cloud-fog tiers. INDICES decides the best cloudlet 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 complementary to INDICES' and benefits from INDICES to make the initial server selection in the fog layer.

VI. Conclusion

Despite the promise of fog/edge computing, user mobility brings in new challenges. Executing a service exclusively on edge devices or fog resources is not acceptable, and choosing the fog server with minimal interference between co-located tasks becomes critical. 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.

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.

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.

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, and 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 and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[3] W. Kuang, L. E. Brown, and 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, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[5] S. Shekhar, H. A. Aziz, A. Bhattacharjee, A. Gokhale, and 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, Jul. 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, and 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, and 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, May 2017, pp. 75–80.

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

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

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

[12] S. Shekhar, A. D. Chhokra, H. Sun, A. Gokhale, A. Dubey, and X. Koutsoukos, "URMILA: Dynamically Trading-off Fog and Edge Resources for Performance and Mobility-Aware IoT Services," Vanderbilt University Institute for Software Integrated Systems, Nashville, TN, USA, Tech. Rep. ISIS-19-101, Mar. 2019.

[13] K. G. Hooper and H. W. McGee, "Driver Perception-Reaction Time: Are Revisions to Current Specification Values in Order?" Tech. Rep., 1983.

[14] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[15] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè, "Broadband Internet Performance: A View from the Gateway," in *ACM SIGCOMM computer communication review*, vol. 41, no. 4. ACM, 2011, pp. 134–145.

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

[17] M. Karimzadeh, Z. Zhao, L. Hendriks, R. d. O. Schmidt, S. la Fleur, H. van den Berg, A. Pras, T. Braun, and 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.

[18] M. R. Rahimi, N. Venkatasubramanian, and 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.

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