Queec: QoE-aware Edge Computing for IoT Devices under Dynamic Workloads

BORUI LI, WEI DONG, GAOYANG GUAN, and JIADONG ZHANG, Zhejiang University and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
TAO GU, Department of Computing, Macquarie University, Australia
JIAJUN BU and YI GAO, Zhejiang University and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China

Many IoT applications have the requirements of conducting complex IoT events processing (e.g., speech recognition) that are hardly supported by low-end IoT devices due to limited resources. Most existing approaches enable complex IoT event processing on low-end IoT devices by statically allocating tasks to the edge or the cloud. In this article, we present Queec, a QoE-aware edge computing system for complex IoT event processing under dynamic workloads. With Queec, the complex IoT event processing tasks that are relatively computation-intensive for low-end IoT devices can be transparently offloaded to nearby edge nodes at runtime. We formulate the problem of scheduling multi-user tasks to multiple edge nodes as an optimization problem, which minimizes the overall offloading latency of all tasks while avoiding the overloading problem. We implement Queec on low-end IoT devices, edge nodes, and the cloud. We conduct extensive evaluations, and the results show that Queec reduces 56.98% of the offloading latency on average compared with the state-of-the-art under dynamic workloads, while incurring acceptable overhead.

CCS Concepts: • **Networks** → **Mobile networks**; • **Human-centered computing** → *Mobile computing*

Additional Key Words and Phrases: Internet of things, edge computing, offloading

ACM Reference format:
Borui Li, Wei Dong, Gaoyang Guan, Jiadong Zhang, Tao Gu, Jiajun Bu, and Yi Gao. 2021. Queec: QoE-aware Edge Computing for IoT Devices under Dynamic Workloads. *ACM Trans. Sen. Netw.* 17, 3, Article 27 (June 2021), 23 pages.
https://doi.org/10.1145/3442363

This work is supported by the National Science Foundation of China under Grant Nos. 62072396 and 61772465, Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020001, and Australian Research Council (ARC) Discovery Project grant DP190101888.
Authors' addresses: B. Li, W. Dong (corresponding author), G. Guan, and J. Zhang, Zhejiang University and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China, 38th, Zheda Rd., Hangzhou, Zhejiang, 310000; emails: {libr, dongw, guangy, zhangjd}@emnets.org; T. Gu, Department of Computing, Macquarie University, Australia; email: tao.gu@mq.edu.au; J. Bu and Y. Gao, Zhejiang University and Alibaba- Zhejiang University Joint Institute of Frontier Technologies, China, 38th, Zheda Rd., Hangzhou, Zhejiang, 310000; emails: bjj@zju.edu.cn, gaoy@emnets.org.

1550-4859/2021/06-ART27 $15.00
https://doi.org/10.1145/3442363

1 INTRODUCTION

Edge computing [7, 29], a new paradigm to complement cloud computing, has gained great popularity recently. Edge computing enables advanced on-device processing and analytics, pushing computing to the edge of the network. Many researchers have explored the advantages of mobile edge computing [26, 30] and proposed different dynamic task offloading approaches applied to mobile devices. These approaches leverage technologies such as VMs and containers and achieve great success in mobile edge computing.

However, low-end IoT devices (e.g., Arduino Uno/Mega/Due, LinkIt Smart, TelosB motes) may not be supported well to offload complex IoT event processing tasks by using existing approaches due to their strictly limited resources (e.g., ~10 KB of RAM and ~50 KB of Flash) [18]. However, they are still the majority of **commercial off-the-shelf (COTS)** IoT devices and will dominate the IoT market for a few years [12]. Even the state-of-the-art edge computing platforms for IoT applications such as ParaDrop [27] do not support low-end IoT devices well, since it requires the support of Linux container, which is barely applicable for the low-end IoT devices without Linux operating systems [18].

It is crucial to enable edge computing, especially task offloading, in low-end IoT devices to develop IoT applications to reduce the cost and facilitate large-scale deployment. In this article, we propose a novel edge computing system that enables transparently offloading computation-intensive tasks from low-end IoT devices to nearby edge nodes or the cloud at runtime. Developers do not need to bother with the implementation details of edge computing and only focus on developing the application logic. However, turning the above ideas into reality faces two practical challenges.

First, it is difficult to design a common edge computing system that supports task offloading and is suitable for all low-end and high-end IoT devices, edge nodes, and the cloud. The existing offloading approaches leverage cross-platform languages such as **Java Virtual Machine (JVM)** [9, 21] or C# **Common Language Runtime (CLR)** [10] to achieve seamless offloading among heterogeneous devices (i.e., devices with different instruction set architectures). Nevertheless, the low-end IoT devices are constrained in both *computation* and *storage* abilities, since they generally have low-frequency **MCUs (microcontrollers)**, ~10 KB memory, and ~50 KB flash (e.g., Arduino Mega 2560 board only has a 16 MHz MCU, 8 KB RAM, and 256 KB Flash). Hence, most of the low-end IoT devices could not support the cross-platform languages or exhibit much overhead to execute them, let alone the migration with these techniques.

Second, the state-of-the-art QoE-aware mobile edge computing framework, MobiQoR [26], proposes a novel framework to improve service response time by relaxing the QoE (e.g., service accuracy). However, the scheduler of MobiQoR overlooks workloads at edge nodes, which may lead to overloading that causes high offloading latency and the multi-threaded execution ability of multi-core processors, which may significantly reduce the offloading latency. There exist several offloading algorithms [3, 4] that consider the existing workloads. Nevertheless, the existing approaches are not applicable in our scenario mainly for two reasons: (1) Existing works only model the relationship between existing workloads and the execution time. Nevertheless, Queec jointly takes the existing workloads, the additional workloads introduced by the new offloaded tasks, and the QoE of new offloaded tasks into consideration to model the execution time when making offloading decisions. This is a novel problem and has not been investigated. (2) Existing works leverage computation-intensive algorithms such as Reinforcement Learning [4] for modeling each task's workload, introducing a non-negligible overhead when making offloading decisions. This overhead is not acceptable, since Queec allows the simultaneous allocation of multiple tasks. Hence, we must build an efficient approach to model

the influence of both the existing workloads and the new workloads introduced by QoE-aware offloading.

To address these problems, we present Queec, a QoE-aware edge computing system for complex IoT event processing under dynamic workloads. First, we carry out a preliminary investigation on where we should place the scheduler, i.e., the cloud, the IoT device, or the edge node. The result shows that scheduling at the gateway edge node can achieve global optimization with the minimal communication overhead and reduces information redundancy. Then, we carefully design Queec's system architecture and the workflow to reduce the overhead of the low-end IoT devices and the edge nodes, especially the gateway. Based on the **RPC (Remote Procedure Call)** offloading, Queec can build more lightweight IoT applications for low-end IoT devices compared with the existing cross-platform migration approaches that leverage heavy-weight virtual machine migration techniques such as References [9, 10, 21], and allow edge nodes to download the required offloading libraries on-demand at runtime. Combined with the TinyLink [17], Queec facilitates the rapid development of IoT applications with edge computing.

Compared with the state-of-the-art QoE-aware offloading approach, MobiQoR [26], Queec (1) takes the existing workloads and multi-threading support on the edge device into consideration when making offloading decisions. The former will lead to superior offloading task allocation, and the latter would utilize the full computing ability of the edge devices. Moreover, (2) superior to MobiQoR, Queec proposes a universal programming model for rapid developing QoE-aware offloading applications. The programming support includes the library templates for developers to implement new offloading APIs (Section 4.1) and the fully implemented RPC-based framework for QoE-aware offloading (Section 4.2).

Queec presents a novel QoE-aware multi-user scheduling algorithm to minimize the overall offloading latency of all tasks. Given the required QoE specified by end-users, Queec ensures that all offloading tasks satisfy these requirements. Queec avoids overloading by gathering the dynamic workloads on edge nodes and formulating them as the constraints of the scheduling algorithm. Queec also fully utilizes the spare computation resources of multi-core processors on edge nodes. With the execution model, which takes the length, size, and resolution of IoT events as input, Queec estimates the execution time of each task offloaded to different edge nodes.

We implement Queec on heterogeneous computational nodes, including IoT devices, edge nodes, and the cloud. We evaluate Queec by building two real-world IoT applications running on both low-end and high-end IoT devices. Results show that:

(1) Queec reduces 56.98% of offloading latency on average under dynamic workloads compared with MobiQoR.
(2) Queec's scheduling algorithm incurs acceptable overhead on the gateway edge node in terms of execution overhead, latency, and throughput.

The contributions of this work are summarized as follows:

- We present Queec, a QoE-aware edge computing system for complex IoT event processing under dynamic workloads. With Queec, the complex IoT event processing tasks that are computation-intensive, especially for low-end IoT devices, can be transparently offloaded to nearby edge nodes or the cloud at runtime.
- We formulate the multi-user scheduling as an optimization problem, which minimizes the overall offloading latency of all tasks. Due to the consideration of dynamic workloads and multi-thread execution capability of edge nodes, Queec schedules multiple tasks in a more accurate and efficient way than MobiQoR.

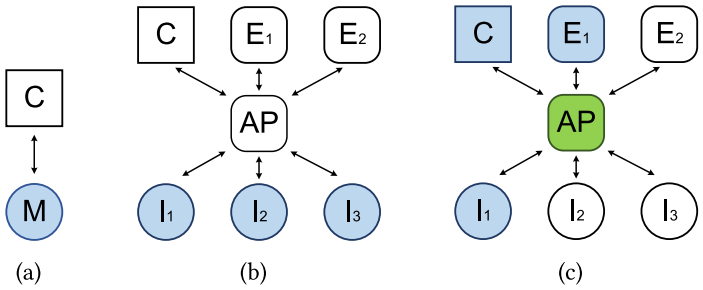


Fig. 1. Topology of the offloading scenario.

- We implement Queec on heterogeneous computational nodes and develop two real-world IoT applications. We conduct extensive evaluations and the results show that Queec reduces offloading latency under dynamic workloads, while incurring acceptable overhead on the gateway edge node.

Section 2 describes our preliminary study. Section 3 introduces Queec through a use case study. Section 4 describes our design considerations and Queec's workflow. Section 5 presents Queec's scheduling algorithm. Section 6 illustrates Queec's implementations, and Section 7 evaluates the performance and the overhead. Section 8 presents the related work, and Section 9 concludes the article.

2 PRELIMINARY STUDY

Several mobile offloading systems [9, 10, 14, 16, 21, 23, 26, 30] have been proposed to offload computation-intensive tasks to the server to reduce the application response time or reduce the battery drain. We intend to shift offloading to edge computing. In this section, we carry out a preliminary study and summarize the main challenges.

2.1 Where to Place the Scheduler

The first challenge is where to execute the scheduling algorithm. Existing mobile systems [9, 10, 14] involve two kinds of participants, the mobile device (M in Figure 1(a)) and the cloud (C in Figure 1(a)). They usually run a scheduler at the mobile device. However, in the scenario of edge computing, two kinds of participants exist, i.e., IoT devices (I_1, I_2, and I_3) and edge nodes (AP, E_1, and E_2) in Figure 1(b). AP represents WiFi AP, which is the gateway edge node in the topology. This not only increases the number and the type of participants, but also changes the network topology.

One solution is so-called *distributed scheduling*, which is to directly migrate the scheduling algorithms from existing work [9, 10, 14, 26] to each IoT device. As shown in Figure 1(b), I_1, I_2, and I_3 perform scheduling locally. However, two issues arise:

- Each distributed scheduler only generates its optimal solution locally. For example, suppose E_2 is capable of executing two offloading tasks simultaneously, while E_1 is capable of executing one task only. I_1, I_2, and I_3 will select the same powerful offloader E_2 that may lead to overloading. However, the synchronization of these offloading decisions among all the IoT devices will introduce significant communication overhead that may cause severe congestion in the network.
- It causes large information redundancy, since all schedulers need a copy of all information.

Our solution is to appoint a participant as the *centralized scheduler* to achieve global optimization and also reduce the redundancy. In Figure 1(c), we have four candidates, i.e., C, I_1,

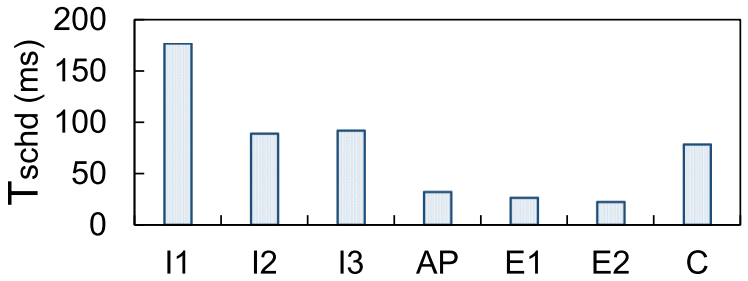


Fig. 2. Overall scheduling latency of each participant as the centralized scheduler.

Table 1. Hardware Specifications of Computational Nodes

Role	ID	Name	CPU	RAM	Storage	Communication
IoT device	I_1	LinkIt Smart 7688	MIPS 24KEc 580 MHz, 1 core	128 MB	32 MB Flash	802.11 b/g/n WiFi
	I_2, I_3	Raspberry Pi 2	ARM Cortex-A7 900 MHz, 4 cores	1 GB	4 GB Flash	802.11 b/g/n WiFi
Edge node	AP	Linksys WRT1900ACS	Marvell 88F6820-A0 C160 1.6 GHz (Armada 385), 2 cores	512 MB	128 MB Flash	802.11 b/g/n/ac WiFi, 1 Gbps Ethernet
	E_1	ThinkPad E431	Intel Core i5-3210M 2.50 GHz, 4C4T*	8 GB	50 GB HDD	1 Gbps Ethernet
	E_2	Dell OptiPlex 7050	Intel Core i7-6700 3.40 GHz, 4C8T*	12 GB	100 GB HDD	1 Gbps Ethernet
Cloud	C	Aliyun ECS hfg5	Intel Xeon Gold 6149 3.10 GHz, 4C8T*	16 GB	40 GB SSD	1 Gbps Ethernet

* 4C4T means the CPU contains 4 cores and could concurrently run 4 threads.

Table 2. Evaluated Execution Time and Latency

ID	I_1	I_2	I_3	AP	E_1	E_2	C
Execution time (ms)	141.65	58.61	58.92	19.76	8.65	3.45	4.03
Latency (ms)	4.124	2.905	3.628	0.047	0.237	0.804	28.359

E_1, and AP. The optimization goal is to minimize the scheduling latency, which is composed of the request round-trip-time T_{req} between devices and the scheduler, and the execution time T_{schd_exec} of the scheduling algorithm. The overall scheduling latency T_{schd} is measured as $T_{schd} = \max\{T_{req} + T_{schd\_exec}\}$.

To better evaluate this issue, we set up an experiment with the topology shown in Figure 1(c). Table 1 shows the hardware specifications. The hardware specification of Aliyun ECS hfg5 can be found in Reference [2]. To estimate T_{schd_exec}, we implement a linear programming problem, similar to Reference [26], with 131 constraints and use the widely used lp_solver [6]. Table 2 shows T_{schd_exec} and T_{req} of each participant. Li et al. [25] reported that the average world-wide RTT to the optimal Amazon EC2 instance is about 74 ms, while Lai et al. [22] reported that the average in-lab RTT to the edge is about 3 ms. Table 2 shows similar results.

To select the centralized scheduler, we evaluate all the participants in Figure 1(c) and search for the one with the minimum overall scheduling latency. It is hard to model the offloading request frequency of various IoT applications due to diversity. For simplicity, we let all the three IoT devices send an offloading request to the scheduler at the same time. Figure 2 shows the overall scheduling latency of each participant as the centralized scheduler. We finally select AP as the centralized scheduler, because it achieves much lower overall scheduling latency than IoT devices and the communication overhead can be greatly reduced. As the scheduler and participants change, we need to alter the scheduling workflow and the scheduling algorithm. We will describe the scheduling design in Section 4.

2.2 Exploring Architecture Heterogeneity

Another challenge is how to offload tasks across different architectures, e.g., offload tasks from ARM on IoT devices to x86 on the server. There exist generally three kinds of approaches to solve the architecture heterogeneity problem:

(1) Most of the mobile offloading systems [9, 10, 16, 21, 24] execute the application code on VMs to hide the heterogeneity, e.g., Dalvik VM of Android on smartphones and Android x86 on servers, and Microsoft .NET Common Language Runtime.
(2) The work in Reference [30] implements emulators on the offloader to dynamically translate the request device's native binary to offloader's native binary and executes the task.
(3) Some mobile offloading systems [14, 23] recompile the mobile applications or their libraries for all target architectures so each platform runs its own native binary.

However, in the edge computing scenario, these approaches are not applicable for the following reasons:

(1) To the best of our knowledge, there is no unified VMs or containers that can run on most of the existing IoT devices, especially resource-constrained low-end IoT devices that do not have traditional OS like Linux.
(2) Each architecture pair of the request device and the offloader, e.g., (STM32, x86) and (AVR, MIPS), demands for a dedicated dynamic translator, which is costly and impractical to implement and deploy.

Inspired by References [14, 17, 23], we intend to solve the heterogeneity problem by offloading method-level tasks and compiling native binary for each API on each platform. However, we are facing the following challenge: To deal with various kinds of offloading tasks, an edge node has to prepare all kinds of native binary in its storage though most of them are hardly used. This introduces huge space overhead for resource-constrained edge nodes, e.g., Linksys WRT1900ACS only has a flash of 128 MB. In Section 4, we will describe how Queec solves this problem.

3 USE CASE STUDY

In this section, we first design two common IoT applications using Queec. We then illustrate the process of application development by going through one of the applications. In the end, we highlight the features that Queec intends to provide.

3.1 Sample Application

We design two sample IoT applications to describe Queec use case. Since speech recognition and face recognition are widely used in IoT community, we design two recognition applications. Voice-controlled LED lamp in Reference [17] is an application that automatically recognizes the user's speech and controls the LED lamp accordingly. We design an edge-computing enabled speech recognition application that can transparently offload the speech recognition task to nearby edge nodes or the cloud. It records the 16 kHz audio as input and translates it into plain text via PocketSphinx [19]. The recognition algorithm is based on the acoustic model that uses Hidden Markov Models with a 5-state Bakis topology. We use the bigram statistical language model from the official release [28]. Face recognition application is another application that takes a photo and recognizes the person in it. The main function is to extract a grey-scaled cropped image of human face and report the confidence level for each recognition process. The recognition task can be offloaded transparently to nearby edge nodes or the cloud.

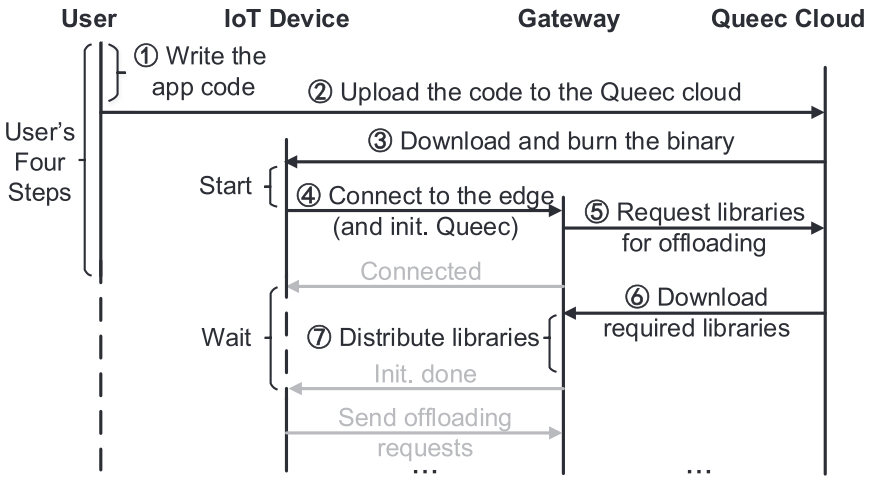


Fig. 3. User workflow and device initialization.

```
1 bool result = Queec_voice.record_to_file("/tmp/voice_01.mp3", 16000,
      3000);
2 if (!result) {
3     bool rtn = Queec_voice.recognize_from_file(filepath, result,
          hmm_model, language_model, dictionary); //speech to text
          recognition
4     if (!rtn && result.find("turn on the light") != string::npos) {
5         ... //corresponding controlling commands
6     }
7 }
```

Fig. 4. Speech recognition application code skeleton.

3.2 Development Process

We illustrate the process of application development with Queec by going through the speech recognition application. Developers only need to follow the first four steps shown in Figure 3. Queec adopts TinyLink's programming style [17], which provides easy-to-use APIs to offload computation-intensive tasks. Developers only need to do the following steps:

(1) Write the application code.
(2) Upload the code to the Queec cloud.
(3) Download and burn the binary to the device.
(4) Connect the device to the WiFi AP.

Developers need not write any annotations about offloading. Figure 4 shows the code skeleton. At line 1, it records a three-second audio file with a sampling rate of 16 kHz. Then it will try to recognize the text with input models, as shown at line 3. Once connected, the IoT device will send the initialization message to the AP to register the device.

Nevertheless, not only IoT devices but also some edge nodes could have limited amount of storage resources. This implies that Queec has to be lightweight so it is able to support resource-constrained devices. Hence, during the initialization phase, edge devices will download the required libraries from the Queec cloud and distribute the libraries to the corresponding devices, which reduces the storage space needed for saving pre-built binaries of various hardware architectures in the edge device. Therefore, after initialization, Queec clients for IoT devices and

edge nodes only contain the basic framework and specific libraries used by its applications. For IoT devices and edge nodes with limited storage, Queec clients periodically run a **least recently used (LRU)** algorithm to maintain a small core of offloading libraries.

3.3 Design Goal

In this section, we describe the three design goals of Queec.

Easy programming. Developer needs not write any annotations about which part to be offloaded, or where and how to offload. Queec adopts TinyLink's programming style [17], which provides easy-to-use APIs to offload computation-intensive tasks.

Transparent offloading. The computation-intensive tasks of complex event processing can be transparently offloaded to powerful edge nodes or the cloud at runtime. Developers write the application as if it runs locally.

Multi-user scheduling. There are multiple devices, multiple offloaders, and a centralized scheduler in our scenario. Each device can request one offloading task at a time. The scheduler assigns all tasks to certain offloaders to minimize the offloading latency of all tasks.

4 SYSTEM DESIGN

In this section, we first describe several important design considerations for Queec. Then, we show Queec's architecture with detailed information about several important components. We finally illustrate how Queec works by presenting the workflow of the sample application in Section 3.

4.1 Design Consideration

We first describe several important design considerations, which are critical in achieving our goal and desired features.

How to deal with heterogeneity? As discussed in Section 2.2, Queec delivers method-level offloading tasks and adopts an RPC-based approach to offload highly abstracted unified APIs (e.g., `recognize_from_file` in Figure 4) that implement functionalities of computation-heavy tasks. Queec developers encapsulate APIs of the same functionality into one offloading library, e.g., `Queec_voice` in Figure 4.

How to extend offloading libraries? Extending libraries is very important for RPC-based offloading approaches. Queec heavily relies on existing library source code (e.g., PocketSphinx) and implements glue code to encapsulate them into Queec's libraries (e.g., `Queec_voice`). Indeed, this approach would hamper the scalability, but it enables task offloading on low-end IoT devices, which is not supported by scalable approaches such as VMs and containers. To alleviate the problem, Queec uses crowdsourcing and allows developers to upload their own libraries. Moreover, Queec provides developers with templates that implement the basic interfaces for creating new libraries. For example, we show the template implementation of `Queec_voice` library in Figure 5. Developers could implement the recording and recognize logic in their own code or machine learning models in the `Queec_voice::record_to_file()` and `Queec_voice::recognize_from_file()` functions.

How to quantify the QoE of libraries? Similar to MobiQoR [26], Queec trades off between QoE and the response time (i.e., offloading latency) in edge computing. Unlike MobiQoR, which normalizes the QoE between 0 and 1, Queec uses a vector to represent the QoE, because many QoE metrics from different applications can neither be normalized nor compared with each other; for example, execution time of recognition algorithms, accuracy (percentage of correctly identified) of object recognition algorithms, normalized mean error of prediction algorithms, resolution and frame rate of video games, and so on. Currently, Queec's QoE vector can be represented as $[exec\_time, accuracy, error, resolution, frame\_rate]$.

```
1  class Queec_voice{
2      public:
3          bool record_to_file(std::string file, int freq, int length);
4          bool recognize_from_file(std::string filepath, std::string
               result, std::string hmm_model, std::string language_model,
                std::string dictionary);
5      ...  //Other variables and functions in template
6  };
7  bool Queec_voice::record_to_file(std::string file, int freq, int
       length){
8      ...  //Implementation
9  }
10 bool Queec_voice::recognize_from_file(std::string filepath, std::
       string result, std::string hmm_model, std::string language_model,
       std::string dictionary)
11     ...  //Implementation
12 }
```

Fig. 5. Example Queec library implementation of `Queec_voice`.

However, it is not always true that larger QoE value means better, e.g., smaller execution time is preferred when accuracy is high. Thus, each element of the vector has a sign (i.e., $-1/1$), indicating whether bigger (i.e., 1) or smaller (i.e., -1) value is better. For example, the QoE vector for a face recognition library can be represented as $[(100, -1), (0.9, 1), (N/A), (N/A), (N/A)]$, which means it can identify a face within 100 ms with an accuracy of 90%.

For the comparison between QoE vectors in Queec, suppose a vector A is pareto-superior than vector B (i.e., all the dimensions of A is superior than B), then we have $QoE_A > QoE_B$ in Queec. Nevertheless, there may exist situations that only 3 out of 5 dimensions of A is superior than B. Towards this situation, Queec brings developers' opinion into consideration to choose the superior one.

How to implement libraries with different QoE? Queec does not set any restriction on how to relax the QoE for libraries. There are multiple ways to relax the QoE such as reducing the amount of reference data, reducing the model complexity (e.g., number of feature vectors [31], model size of DNN [32]), reducing the number of iteration, and so on. Queec library developers should evaluate the QoE vectors and record them in the libraries.

What if the network is down? There exist some extreme cases in practice, e.g., (1) only gateway edge node and IoT devices exists; (2) only one IoT device exists when the local network is down. For the former case, the gateway edge node itself will be the offloader and act as both the scheduler and the only offloader. For the latter case, which is very rare, the IoT devices will log all requests in its storage and wait until the network recovers.

4.2 Architecture and Workflow

Figure 6 shows Queec's overall architecture. Queec provides a lightweight homogeneous architecture for all edge nodes and IoT devices. The profiler monitors the statuses of the network (e.g., bandwidth, latency) and the system (e.g., CPU workload, memory usage). Since low-end IoT devices do not provide system commands to estimate these data, we design and implement a simplified and lightweight profiler. For example, to measure the network latency and bandwidth for Arduino DUE, the profiler sends a fixed 10 KB of data to the AP via TCP connection, which is

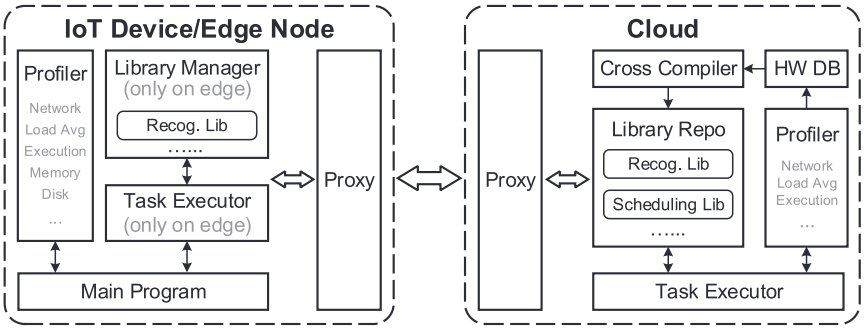


Fig. 6. Queec architecture overview.

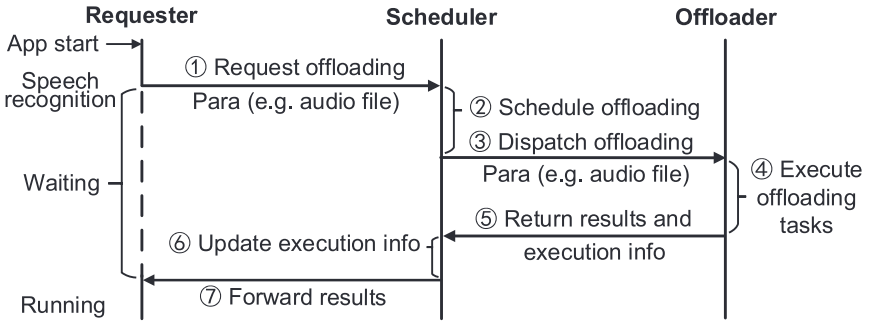


Fig. 7. Workflow illustration.

similar to the approach in Reference [14]. Figure 7 shows the seven steps of a sample workflow, which illustrates how the speech recognition API in Figure 4 is offloaded.

5 QOE-AWARE SCHEDULING

Once receiving offloading requests, the scheduler will search all possible offloaders to execute the requests while maintaining user-required QoE and balancing the tasks among all offloaders. We describe the problem formulation and how to optimize the scheduling in this section.

5.1 Problem Formulation

We first introduce the following notations:

- M, N: M represents the set of all possible offloaders, including all IoT devices, edge nodes, and the cloud. In the example in Section 2, $M = \{I_1, I_2, I_3, E_1, E_2, AP, C\}$. N denotes the set of all tasks to be offloaded.
- i, j, d_{ij}: d_{ij} represents the indicator variable. $d_{ij} = 1$ if offloading task j is designated to the offloader i, where $i \in M,\ j \in N$. Otherwise, $d_{ij} = 0$ indicates offloading task j is not designated to offloader i.
- $Q_{ij}, Q_{ij}^S, Q_j^U : Q_{ij}^S$ denotes the set of all possible QoE vectors of offloading libraries on offloader i for task j. Q_j^U denotes the user required QoE vector of task j. Q_{ij} denotes the chosen QoE vector for offloading task j to offloader i, where $Q_{ij} = \min\{q \geq Q_j^U\},\ q \in Q_{ij}^S$.
- ω_j, τ_{ij}, f: ω_j denotes the quantity vector (e.g., $[size, duration, resolution]$) of task j, and τ_{ij} denotes the execution time of task j designated to offloader i. We use trained model $f()$ to estimate execution time τ_{ij}. $f()$ is a linear model that takes the quantity vector as input and outputs the task execution time, which will further elaborate in Section 5.3.

- l_{ij}, g, L_i^E, L_i^M: L_i^E represents the existing workloads of offloader i, while L_i^M represents the maximum workloads that offloader i can approximate. Usually, L_i^M equals the number of processor cores. l_{ij} denotes the estimated incremental workloads if task j is offloaded to offloader i. $g()$ denotes the trained model for estimating l_{ij}.
- P_i, R_i, D_j: P_i denotes the throughput between offloader i and the scheduler, R_i denotes the RTT between the scheduler and offloader i. D_j denotes the size of input data to be transferred for offloading.
- T_j^D, T_{ij}^O, S: T_j^D represents the round-trip transmission time of task j between the requester and the scheduler, while T_{ij}^O represents the one between offloader i and the scheduler. S represents the execution time of the scheduling algorithm on the scheduler.

With the above definitions, the scheduler will search all possible offloaders to achieve the *minimal offloading latency*. As there may be multiple offloading requests in the scheduler's receiving queue at a time, we define the offloading latency as the time span from the beginning of the scheduler searching for a set of offloaders to the last arrival of all offloading results. With the workflow in Figure 7, the minimal offloading latency for N offloading tasks can be written as:

$$\min \quad S + \max_{j \in N} \left[\sum_{i \in M} (\tau_{ij} + T_{ij}^O) d_{ij} + \frac{1}{2} T_j^D \right]. \tag{1}$$

In the above equation, the $\sum_{i \in M} (\tau_{ij} + T_{ij}^O) d_{ij}$ represents the time of estimating execution time, transmitting the task to the offloader and acquire results from the offloader (i.e., steps ②③⑤ in Figure 7), and the $\frac{1}{2} T_j^D$ indicates the time transmitting results to the requester (i.e., step ⑦ in Figure 7), which we do not count the transmission time from the requester to the scheduler, since the tasks have already arrived. We simplify the round-trip transmission time T_{ij}^O as $D_j/P_i + R_i$, because the results of offloading tasks are usually short plain text whose transmission time can be ignored.

This offloader selection problem can be formulated as an optimization problem, with the optimization criterion being minimizing the offloading latency and the three following constraints:

(1) For each task, the QoE of the offloader's library implementation should be greater than the user-required one.
(2) For each offloader, the sum of existing workloads and the workloads introduced by the offloading task should not exceed the offloader's maximum workloads.
(3) Each offloading task should be executed at only one offloader.

This is a typical mini-max problem described in Reference [8]. We add an auxiliary variable y to formulate the optimization problem as follows:

$$\begin{aligned}
\min \quad & S + y \\
\text{s.t.} \quad & y \geq \sum_{i \in M} \left(\tau_{ij} + \frac{D_j}{P_i} + R_i \right) d_{ij} + \frac{1}{2} T_j^D, \quad \forall j \in N \\
& \sum_{i \in M} Q_{ij} d_{ij} \geq Q_j^U, \quad \forall j \in N \\
& L_i^E + \sum_{j \in N} l_{ij} d_{ij} \leq L_i^M, \quad \forall i \in M \\
& \sum_{i \in M} d_{ij} = 1, \quad \forall j \in N \\
\text{var.} \quad & \{d_{ij}, \forall i \in M, \ \forall j \in N\}.
\end{aligned} \tag{2}$$

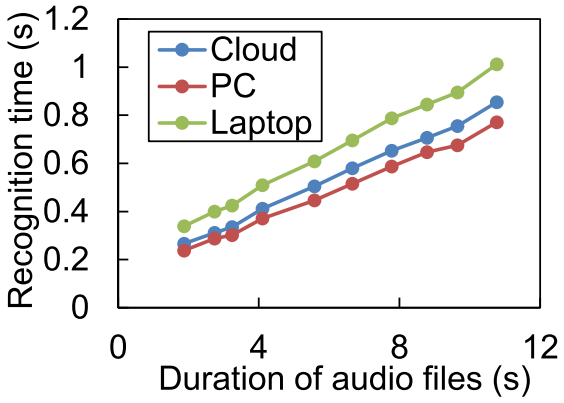


Fig. 8. Recognition time under different duration.

This is the **mixed integer linear programming (MILP)** problem, which is NP hard. However, considering the scale of wireless local area network is not large, approximately from 10 to 100 computation nodes, it is acceptable to directly use general ILP solvers to resolve the problem. We will present how Queec estimates the execution time τ_{ij} and the load average l_{ij} in the following sections.

5.2 Comparison with MobiQoR

MobiQoR [26] considers a mobile edge network with a client device and M edge nodes. MobiQoR divides the request with total workload (e.g., a gallery of 20 photos in MobiQoR's evaluation) into N pieces and distribute them to the edge nodes. It tries to minimize the response time and the energy consumption of *all N nodes*, while Queec tries to minimize the response time of *all offloading tasks*. With the notations in Queec, we formulate MobiQoR's main scheduling algorithm as follows:

$$\begin{aligned} \min \quad & \max_{i \in M} \left[\sum_{j \in N} \left(\frac{D_j}{P_i} + R_i + \tau_{ij} \right) d_{ij} \right] \\ \text{s.t.} \quad & \sum_{i \in M} d_{ij} = 1, \quad \forall j \in N. \end{aligned} \tag{3}$$

Queec can generate better solutions than MobiQoR for two reasons:

(1) MobiQoR overlooks the multi-threaded execution, because it calculates the sum of multiple tasks' execution time as the offloader's overall execution time, while Queec captures the capability of multi-cores and can greatly reduce execution time of multiple tasks.
(2) MobiQoR overlooks the dynamic workloads on offloaders, which may greatly affect the execution time, while Queec models the influence of existing workloads on offloaders and can avoid overloading.

5.3 Estimation of Execution Time and Workloads

Execution time estimation is very difficult, since there are so many influencing factors that we can only consider several key factors. MobiQoR considers that the execution time is in the linear relationship with the workload (i.e., task number). However, it overlooks a very important feature, the *quantity* of the complex IoT events, e.g., the duration of an audio file, the resolution of an image, and so on. To estimate their relationship, we conduct an experiment using speech recognition. We recognize 10 audio files on the laptop, the desktop PC, and the cloud server, as illustrated in Table 1, using a library whose QoE is about 81%. In Figure 8, we observe that the execution time increases

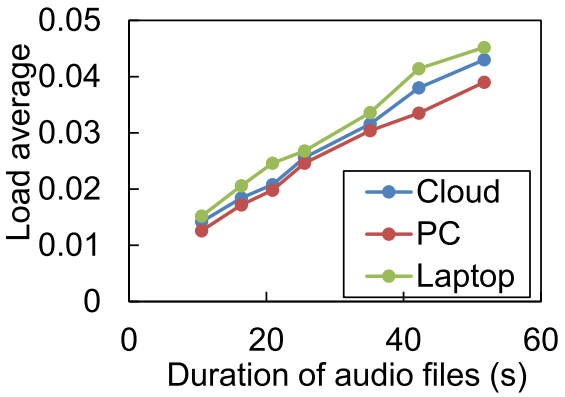


Fig. 9. Workloads under different duration.

almost linearly when the duration increases. We intend to use a linear model $f_{(i,Q_{ij},L_j^E)}$ to estimate the execution time τ for the library with a QoE of Q_{ij} at the offloader i:

$$\tau_{ij} = f_{\left(i,Q_{ij},L_j^E\right)}(\omega_j) = \alpha_{ij}^{\tau}\omega_j + \beta_{ij}^{\tau}, \quad \forall i \in M, \forall j \in N, \tag{4}$$

where the coefficients α_{ij}^{τ} and β_{ij}^{τ} can be trained offline and stored in the scheduler's database for future estimation.

We try to measure workloads by using a widely used metric in Linux system, the load average, because it can reveal the average dynamic workloads in a period of time, e.g., 1 minute. Then, we estimate the relationship between introduced workloads and the task quantity via an experiment as before. Figure 9 shows the result. Similarly, the task workload can be formulated as:

$$l_{ij} = g_{(i,Q_{ij})}(\omega_j) = \alpha_{ij}^{l}\omega_j + \beta_{ij}^{l}, \quad \forall i \in M, \ \forall j \in N, \tag{5}$$

where the coefficients α_{ij}^{l} and β_{ij}^{l} can be trained offline. Note that if the execution time and workloads are *not linear* with the quantity of the input, then Queec estimates the non-linear results offline and stores the relationship in tables for quick searches.

6 IMPLEMENTION

In this section, we discuss the implementation details of Queec prototype.

Library. Currently, we have implemented two real-world libraries: the voice recognition library and the face recognition library. We encapsulate it as dynamical linking libraries `.so` file for Linux distributions and OpenWrt standard packages `.ipk` for OpenWrt systems. We use CMUSphinx to implement the voice recognition function on the cloud server and use PocketSphinx [19] to implement for edge nodes and IoT devices. The QoE value of the implemented library are 98%, 94%, 73%, and 42% for the cloud, the laptop, the Raspberry Pi 2, and the Linksys WRT1900ACS, respectively, by reducing the size of the trained language model and the dictionary.

Proxy. We use the Google's Protocol Buffers 2 as the serialization tool on all computation nodes, and nanopb, a lightweight version, for low-end IoT devices. There are four types of message in the `.proto` file, which are heartbeat beacons, offloading requests, offloading responses, and errors. We also provide implementations of MQTT Brokers (e.g., Mosquitto) and clients (e.g., Paho) for IoT communication.

Task Executor. The task executer is responsible for handling the tasks offloaded to the device. To fully utilize the computation resources of multi-core processors on edge and cloud devices, the task executor allocates each offloaded task to a dedicated process, which could be scheduled to a particular CPU core. With this mechanism, the multi-core processors of both edge and cloud

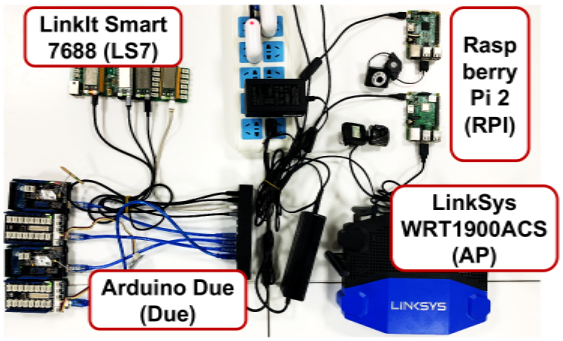


Fig. 10. Computational nodes testbed of Queec. The PC and laptop are wirelessly connected to the AP and are not shown in this picture.

devices could be fully utilized, which could achieve better performance than only use an omni-process like the MobiQoR [26] does.

Scheduler. As the formulated problem is a typical mixed integer linear programming problem, we utilize the widely used ILP solver, lp_solve [6], to develop the scheduler. It is worth noting that when we cross compile the source code of the latest version `5.5.2.5`, we find that the compiled library will generate undiscovered wrong solutions. When we change back to version `5.5.2.0`, the cross compiled library performs correctly again.

7 EVALUATION

In this section, we evaluate the Queec's performance and overhead from different perspectives.

7.1 Experiment Setup

Computational node. We implement Queec on all computational nodes, as illustrated in Table 1, including four LinkIt Smart 7688 (LS7 for short), two Raspberry Pi 2 (RP2 for short), a Linksys WRT1900ACS (AP for short), a personal laptop (Laptop for short), a desktop PC (PC for short), and an Aliyun cloud server (Cloud for short). In addition, we also implement four low-end IoT devices using Arduino Due (Due for short) with WiFi connectivity, each equipped with a Waveshare Music Shield and an OpenJumper Camera module. LS7 and RP2 are equipped with USB Webcams and USB Microphones. Our testbed of computational nodes (except PC and Laptop) is illustrated in Figure 10, and the PC and the Laptop are wirelessly connected to the AP. The laptop, DELL PC, and the cloud are the offloaders, and all the IoT devices are the requesters. The network topology is similar to Figure 1(c).

Application and library. We use the two sample applications illustrated in Section 3 for evaluation, because they are widely used in IoT communities. We have encapsulated dynamical linking libraries by using OpenCV 3.3.0 and PocketSphinx for evaluation, and other sophisticated recognition techniques can also be applied. As discussed in Reference [26], there exist many approaches to achieve different QoE implementations. We use dictionary sizes to control the speech recognition QoE and use synthetic voice for recognition. Similarly, we reduce the dimension number of feature vectors for face recognition and use photos from the AT&T Face database [1].

7.2 Performance

Offloading latency. We first have a detailed look at the overall offloading latency of both MobiQoR and Queec when there are different workloads on offloaders. We let two LS7 and two RP2 send their offloading requests to the scheduler simultaneously, which are the four tasks. We set the

Table 3. Relaxing QoE (i.e., Accuracy) by Reducing Reference Word Number in Dictionary for Speech Recognition

# of words	300	1000	3000	10000	33672	134723
Dict size (KB)	6.1	25.3	73.5	240.0	792.5	3195.4
QoE	73.17%	76.62%	81.22%	85.70%	91.46%	95.12%

Table 4. Relaxing QoE (i.e., Accuracy) by Reducing Dimension Number for Face Recognition Using PCA (Eigenfaces)

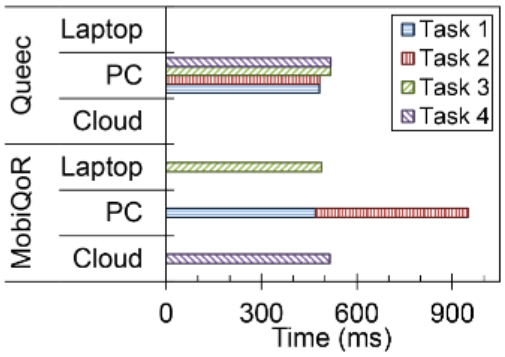
Dimension number	10	20	30	50	100
Model size (MB)	2.9	5.5	6.8	6.8	6.8
QoE	84%	92%	92%	92%	92%

powerful offloader, PC, to few workloads (i.e., workload=2), full workloads (i.e., workload=8), and overloaded condition (i.e., workload=16) before scheduling by running 2, 8, and 16 π calculation applications, respectively, which use the widely used tool bc on Linux systems. We use the 3.24 s audio file and 112x92 photos as input and set the QoE to 80% and 90% for the speech recognition and the face recognition, respectively. Queec's scheduler chooses the speech recognition library whose QoE is 81.22%, as illustrated in Table 3, and the face recognition library whose QoE is 92% and dimension number is 30, as illustrated in Table 4.

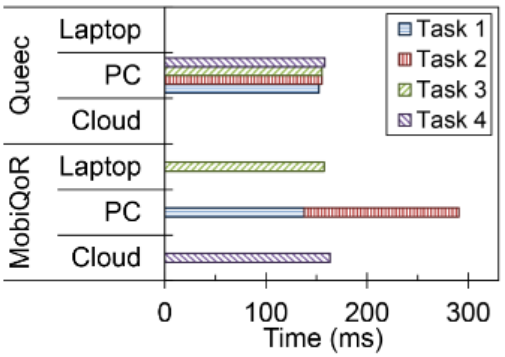
Figure 11 shows the result. It is easy to observe that Queec reduces at least 46.39% of the overall offloading latency than MobiQoR in six cases. This is because MobiQoR's scheduler tries to distribute offloading tasks to every offloader without considering their multi-threaded execution ability nor the workloads on offloaders. As shown in Figures 11(c) and (d), when the powerful offloader is overloaded, MobiQoR's scheduler still delivers two tasks to it, which causes a dramatic increase in the offloading latency. When the PC edge device workload (i.e., 16) dramatically exceeds the maximum workloads (i.e., 8), the offloading decisions of Queec and MobiQoR for workload=8 and workload=16 are the same while the performance (i.e., overall execution latency) of MobiQoR further worsened, since it still allocates tasks to the overloaded PC, as shown in Figures 11(e) and (f). On the contrary, Queec avoids delivering offloading tasks to overloaded ones and fully utilizes offloaders' multi-core processors. Note that in Figures 11(c), (d), (e), and (f), Queec's scheduler delivers offloading tasks to different offloaders, because the latency reduction of cloud offloading is less than the latency increase caused by transmission.

Then, we evaluate the overall offloading latency of two recognition applications under different existing workloads on offloaders. This time, we use four low-end IoT devices, i.e., two Due and two LS7, and let each continuously send one offloading request to the scheduler simultaneously with a period of every five seconds. Figures 12(a) and (b) show the results. We observe that Queec reduces 63.69% and 65.54% of the overall scheduling latency for speech recognition and face recognition, respectively. The more existing workloads on powerful offloaders, the more overall latency it introduces by MobiQoR's scheduling. The main cause is still that MobiQoR overlooks existing workloads on offloaders and the multi-thread execution ability.

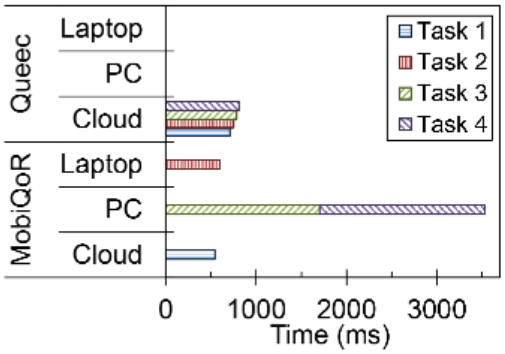
Breakdown of response time. Then, we take an in-depth look at the response time of the application's requests by using Queec. We divide response time into seven parts, as illustrated in Figure 7. We add one additional separation into the execution of offloading task, to watch the preparation time for offloading, including the time for locating the library and dynamically loading it into Queec. Figure 13 shows the breakdown of response time for the two applications. We observe that:



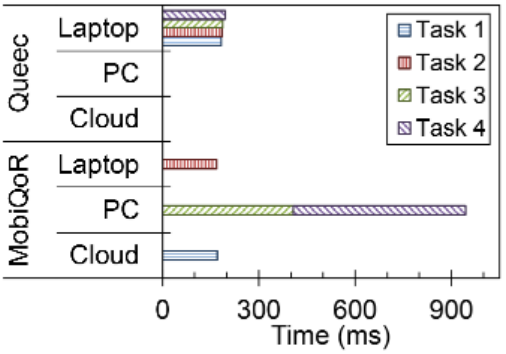
(a) Speech recognition (PC workload=2).



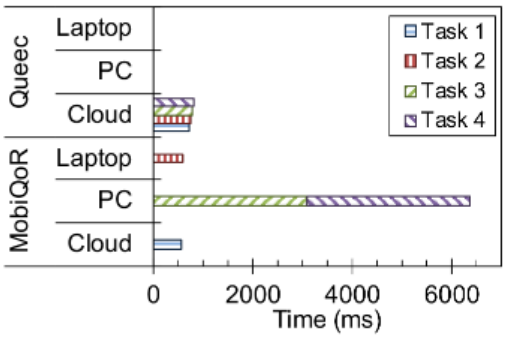
(b) Face recognition (PC workload=2).



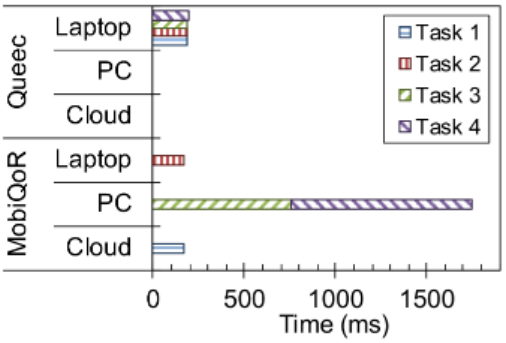
(c) Speech recognition (PC workload=8).



(d) Face recognition (PC workload=8).

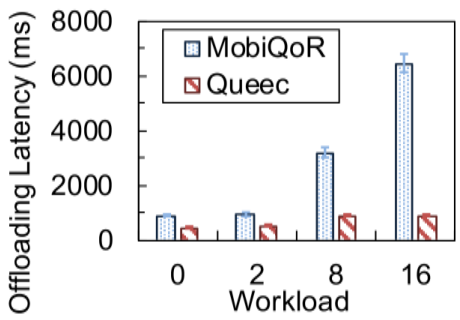


(e) Speech recognition (PC workload=16).

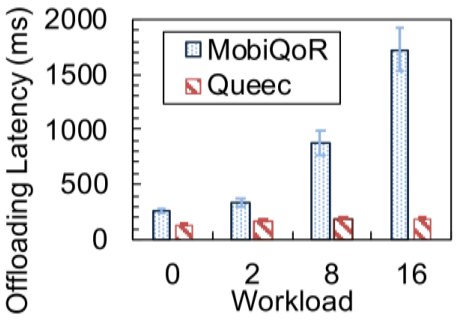


(f) Face recognition (PC workload=16).

Fig. 11. Detailed offloading latency of MobiQoR and Queec under different workloads on the PC edge device.



(a) Speech recognition.



(b) Face recognition.

Fig. 12. Overall offloading latency of two recognition applications under different workloads on PC edge device.

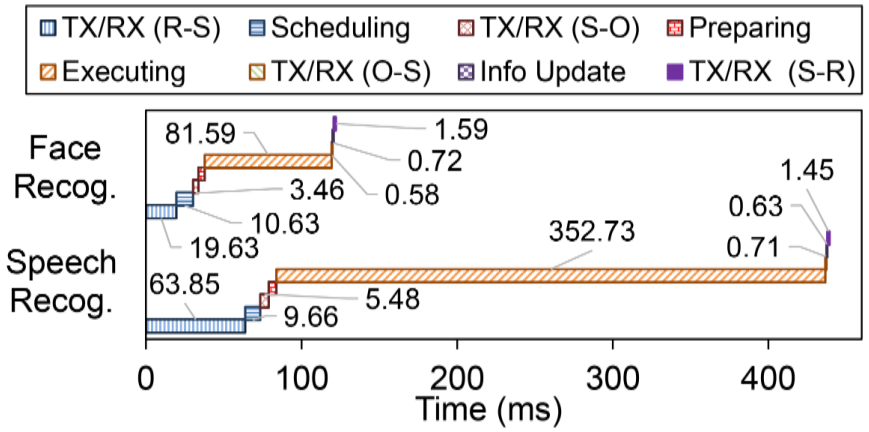
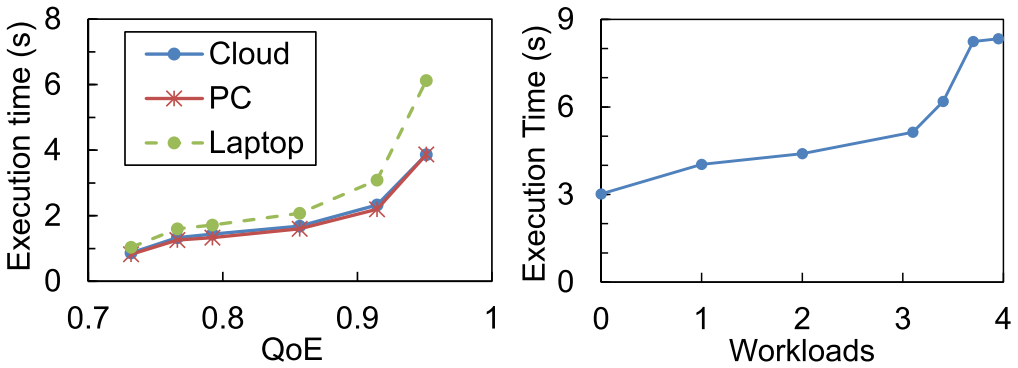


Fig. 13. Breakdown of response time.



(a) Under different QoE. (b) Under dynamic workloads.

Fig. 14. Influence of task QoE and offloader's dynamic workloads on execution time of offloading tasks.

(1) For both applications, the execution time of the offloading task is the main source of the response time, about 80.28% and 66.78%, respectively.
(2) Excluding the transmission time and the offloading execution time, Queec's fundamental procedure involves small execution time, about 15 ms on average.
(3) The transmission time from the request to the scheduler (i.e., TX/RX (R-S)) is usually much larger than the one (i.e., TX/RX (S-R)) back from the scheduler to the requester.

This is because the transmission request usually contains large input data such as audio files. Also the return trip does not perform a TCP three-way handshake, since the client is waiting and the connection is kept alive.

7.3 Microbenchmark

Lines of code. Queec's easy programming feature enables developers to rapidly build IoT applications. We intend to compare the lines of code needed to implement the sample applications between executing locally, offloading using original APIs and offloading using Queec. Table 5 shows the result. Using Queec can reduce the lines of code by 75.20% to 94.87%. This is because Queec provides developers with easy-to-use APIs to implement offloading functionalities.

Execution time. We intend to evaluate the relationship between the QoE and the execution time by testing speech recognition. The QoE of libraries vary from 73.17% to 95.12%, as illustrated in Table 3. The input is a 20-second audio file with a size of 642 KB. Figure 14(a) shows the result and we observe that on the same platform, the execution time is in almost linear relationship

Table 5. Lines of Code for Implementing Applications

Application	Executing locally	Offloading via original APIs	Offloading via Queec
Speech Recognition	571	916	47
Face Recognition	254	729	63

Table 6. Execution Time of Speech Recognition on different Computational Devices

Word count	Duration of audio (s)	Execution time (s)				
		Cloud	PC	Laptop	RP2	LS7
5	1.87	0.27	0.24	0.34	3.08	18.69
8	3.24	0.33	0.30	0.43	4.36	21.68
13	5.58	0.50	0.45	0.61	7.15	27.92
16	6.66	0.58	0.52	0.70	8.43	30.88
20	7.78	0.65	0.59	0.79	9.75	34.09

with the QoE, and increases dramatically when the QoE exceeds 91.46%. This phenomenon is also reported in MobiQoR.

Similarly, we intend to evaluate the influence of existing workloads on the execution time. We continuously offload speech recognition tasks to an RP2 using the 3.24 s audio file with a QoE of 81.22%, while increasing the workloads on it by running multiple computation-intensive tasks. Since the RP2 has a quad-core processor, its maximum workload can reach up to four. Figure 14(b) shows that the execution time increases linearly as the existing workloads go up, since there is still one available core for execution. When the workload is above the maximum workload minus one, execution time increases dramatically—even doubles due to competitions among computation resources. When the existing load average is below the maximum load average minus one, the execution time increases slightly as the load average goes up, because there is still an empty processor core available for offloading. As illustrated in Section 5.3, our estimation model captures the existing load average and can estimate the execution time more accurately.

First, we will evaluate whether the relationship between the execution time of the offloading task and the quantity of the task (i.e., duration of audio file) is linear in both speech recognition and face recognition scenarios. Suppose the user required QoE for speech recognition is 80%, and we choose the library implementation whose QoE is 81.22% in Table 3. We use the testing sentences described in Section 5.3 for evaluation. Table 6 shows the result. We observe that the PC has the shortest execution time, more than about 50× quicker than the LS7. The Cloud requires about additional 10% of execution time than the PC. We also observe that this task is too computationally heavy for IoT devices, especially LS7, whose execution time is far beyond acceptable.

However, we choose the face recognition library whose QoE is 92% and dimension number is 30, i.e., similar number used for five people in Reference [31]. The data for classification is the same as that described in Section 6. In Table 7, we observe that the execution time of face recognition increases slightly as task quantity increases. We plot the breakdown of the execution time and find out that the little increased time is caused by reading large photos from the storage. The time for scaling down photos to a resolution of 112x92, and the time for recognition stay almost the same on each offloader.

Table 7. Execution Time of Face Recognition on different Computational Devices

Resolution	File size (KB)	Execution time (s)			
		Cloud	PC	Laptop	RP2
112x92	11	0.07	0.06	0.10	1.60
1024x768	769	0.07	0.07	0.11	1.63
4160x3120	12,675	0.13	0.15	0.21	1.92

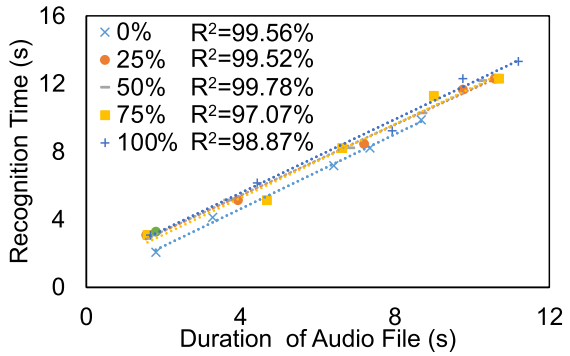


Fig. 15. Execution time of voice recognition under different QoEs.

To evaluate the effectiveness of $f()$ function that we used to estimate the task execution time with quantity vector in Section 5.3, we generate the audio files by translating sentences, whose number of words ranges from 4 to 32, as the evaluation input. We set the QoE as another variable, ranging from 0 to 100% with the step of 25%. Figure 15 shows the result, and the dotted lines illustrate the linear model $f()$ of each QoE. We can observe that the execution time of Queec's implemented voice recognition is approximately a linear function of the quantity of audio files. We use the coefficient of determination (i.e., adjusted R^2 value) of each regression model to depict the effectiveness of our prediction model, which is also illustrated in Figure 15. Results show that the average adjusted R^2 value is 98.96%, which means that the regression model could represent the observed execution time to a large extent.

Load average. We intend to evaluate the influence of existing load on the execution time. We continuously offload speech recognition tasks to an RP2 using the 3.24 s audio file with a QoE of 81.22%, while increasing the load average on it by running multiple computation-intensive tasks. Since the RP2 has a quad-core processor, its maximum load average can reach up to four. The evaluation result in Figure 14(b) shows that the existing load average does affect the execution time. When the existing load average is above the maximum load average minus one, the execution time increases dramatically—even doubles due to competitions among computation resources. When the existing load average is below the maximum load average minus one, the execution time increases slightly as the load average goes up, because there is still an empty processor core available for offloading. As illustrated in Section 5.3, our estimation model captures the existing load average and can estimate the execution time more accurately.

7.4 Overhead

In this section, we intend to find out the influence of scheduling on the WiFi AP. We first evaluate the executing time of scheduling algorithm. We generate a linear programming problem with

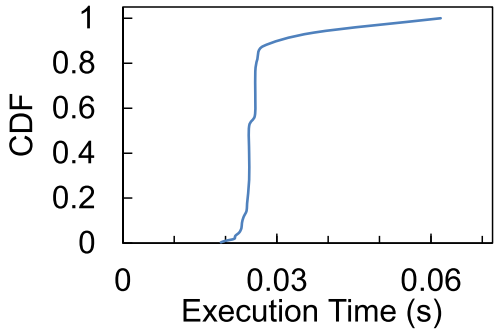


Fig. 16. CDF curve of the scheduling time.

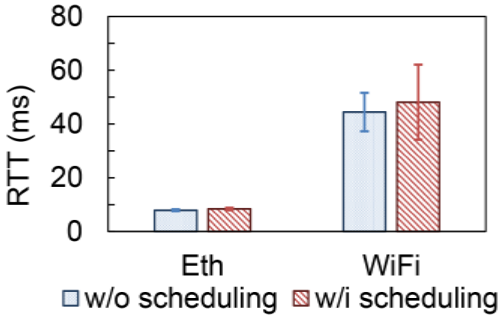


Fig. 17. Scheduling influence on WiFi latency.

131 constraints and run the scheduling algorithm. The result is shown in Figure 16. Through the **cumulative distribution function (CDF)** figure, we can find that around 80% of the execution times are below 26 ms and none of them exceeds 63 ms, which is relatively small. If the scheduler continuously runs the scheduling algorithm, then Queec can schedule about 40 rounds per second.

Next, we evaluate the scheduling overhead introduced by executing the solver continuously without sleeps. We record the RTT of the wired connection between the PC and the cloud, as well as the wireless connection between the RP2 and the cloud. We also evaluate the throughput of the two connections by using `iperf`. During the evaluation, four end-users are connecting to the AP via their smartphones running chat applications. Each time before we conduct the experiments, we synchronize all their clocks by using `ntpdate`. In Figure 17, we observe that the RTT increases slightly, i.e., 0.46 ms and 3.67 ms on average for the wired and the wireless, respectively. In Figure 18, we observe that the throughput drops a little, i.e., about 3.59% and 2.95% of the original throughput on average for the wired and the wireless, respectively. Therefore, considering very little influence on the RTT and throughput, we set the AP to execute its scheduling algorithms continuously without sleeps in our evaluations.

8 RELATED WORK

In this section, we discuss the related work from the following aspects:

Mobile computation offloading. A significant amount of previous work [9, 10, 16, 21, 23, 24, 30] has explored the area of offloading computation intensive tasks from the resource-constrained mobile devices to the cloud to accelerate the task execution or reduce the battery drain. MAUI [10] relies on Microsoft .NET Framework to offload C# code, which has been annotated as remoteable by the user from smartphones running Windows to the cloud at the method-level granularity. CloneCloud [9] seamlessly migrates threads of the application on Android smartphones to the

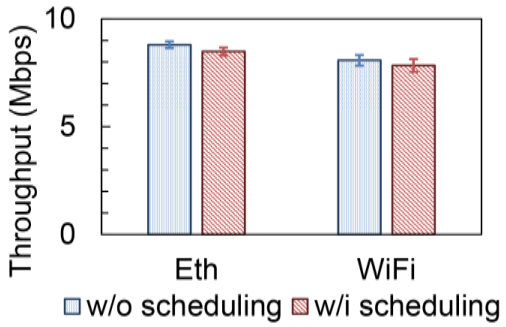


Fig. 18. Scheduling influence on WiFi throughput.

VM-based device clones on the cloud. ThinkAir [21] proposes a new framework, which combines MAUI and CloneCloud by parallel method execution using dynamically requested on-demand VM images to enhance the elasticity and scalability of the cloud.

COMET [16] can also been seen as a combined system of MAUI and CloneCloud implemented with the **distributed shared memory (DSM)** to minimize the communication overhead between smartphones and the server. Lee et al. [24] enhance the DSM model in COMET to improve the efficiency and the reliability for offloading concurrent multiple threads. Georgiev et al. [14] develop LEO, a mobile sensor inference algorithm scheduler designed to maximize the performance of concurrently executing mobile sensor apps without changing inference accuracy, and Dong et al. [11] leverage a graph compression and combining algorithm to obtain the optimal solution for the multi-user offloading problem in the mobile edge computing scenario.

Kim et al. [20] is a signal strength-aware adaptive offloading approach for image processing applications on mobile devices. This paper investigates the relationship between the **received signal strength indicator (RSSI)** and the offloading performance in terms of energy consumption on mobile devices and presents an RSSI-aware offloading algorithm.

Another recent work, Dandelion [15], builds a code offloading system for wearable devices such as Google Glass. Dandelion is designed as a separate component against the wearable applications so it could be a universal service for offloading multiple applications.

Unlike the above mobile offloading systems that involve the mobile devices and the cloud, Queec focuses on offloading scenarios of IoT devices, which brings new challenges and opportunities. Introducing IoT devices not only changes the type of offloading participants, but also changes the problem formulation. Moreover, the heterogeneity problem discussed in Section 2 makes existing mobile offloading systems difficult to be directly applied.

Lee et al. propose Native Offloader [23], which is an automatic cross-architecture computation offloading framework. It selects offloading tasks with offline profiling at compile time and generates offloading-enabled native binaries for both mobile device and server.

Another similar offloading system [30] automatically offloads computation-intensive native binaries to the server at runtime via dynamic binary translation across the device and the server. Similarly, it is not applicable to prepare dynamic binary translators for each architecture pair on the edge nodes.

Recently, with the proliferation of container-based applications, migrating containers among multiple architectures is also important for building applications that exhibits high-efficiency. Hence, H-Container [5] is proposed for tackling this question. H-Container achieves cross-**ISA (Instruction Set Architecture)** execution without the source code at the user space by the executable binary transformation and leverages the **CRIU (Checkpoint Restore In User-space)** mechanism of Linux for seamless container execution migration.

However, with the prior knowledge of underlying client-server architecture pair, e.g., (STM32, x86), (AVR, MIPS), the above approaches generate tightly coupled pair-wise native binaries. Due to the heterogeneity problem in IoT devices, it is costly and impractical to implement specific native binaries for each pair of heterogeneous devices. Unlike these approaches, Queec adopts an RPC-based offloading approach that hides IoT devices' heterogeneity and builds lightweight IoT applications for low-end IoT devices.

Edge computing. Recent years have witnessed the innovations in a new computing paradigm, the edge computing [13, 29] (a.k.a. fog computing [7]). Liu et al. propose and implement a WiFi AP-based edge computing platform, ParaDrop [27], with virtualization techniques. The system supports multi-tenancy of WiFi APs and orchestrates APs through a cloud-based backend. Different from providing services and managing resources among APs in ParaDrop, Queec mainly focuses on implementing a computation offloading system on the edge and targets on offloading.

Li et al. propose a mobile edge computation framework, MobiQoR [26], to trade QoE for reduced latency and extra energy on mobile devices, based on the observation that edge applications can tolerate some level of quality loss. Different from MobiQoR, Queec builds an edge computing system for complex event processing. Also, MobiQoR overlooks existing workloads on edge devices, which may lead to significant performance degradation if choosing an overloaded powerful offloader. Moreover, MobiQoR assumes all offloading workloads are from the same kind of applications, which limits its capability of offloading heterogeneous IoT tasks.

9 CONCLUSION

In this article, we present Queec, a QoE-aware edge computing system for complex IoT event processing under dynamic workloads. With Queec, developers can transparently offload computation-intensive tasks from low-end IoT devices to nearby edge nodes or the cloud. We implement Queec and build two real-world applications on heterogeneous computational nodes. Evaluation results show that Queec reduces 56.98% of the offloading latency on average compared with MobiQoR under dynamic workloads.

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable comments and helpful suggestions.

REFERENCES

[1] Cambridge University Engineering Department. 2020. The AT&T Database of Faces. Retrieved from https://www.kaggle.com/kasikrit/att-database-of-faces.

[2] Alibaba Cloud. 2018. Hardware Specification of Aliyun Elastic Compute Service (ECS) hfg5. Retrieved from https://help.aliyun.com/document_detail/25378.html.

[3] Anil Acharya, Yantian Hou, Ying Mao, Min Xian, and Jiawei Yuan. 2019. Workload-aware task placement in edge-assisted human re-identification. In *Proceedings of the IEEE SECON*.

[4] Adam A. Alli and Muhammad Mahbub Alam. 2019. SecOFF-FCIoT: Machine learning based secure offloading in Fog-Cloud of things for smart city applications. *InternetThings* 7 (2019), 100070.

[5] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. 2020. Edge computing: The case for heterogeneous-ISA container migration. In *Proceedings of the ACM VEE*.

[6] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. 2004. lp_solve 5.5, open source (mixed-integer) linear programming system. Retrieved from http://lpsolve.sourceforge.net/5.5/.

[7] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the ACM MCC Workshop on Mobile Cloud Computing*.

[8] C. Charalambous and A. Conn. 1978. An efficient method to solve the minimax problem directly. *SIAM J. Numer. Anal.* 15, 1 (1978), 162–187.

[9] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. 2011. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the ACM EuroSys*.

[10] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. 2010. MAUI: Making smartphones last longer with code offload. In *Proceedings of the ACM MobiSys*.
[11] Luobing Dong, Meghana N. Satpute, Junyuan Shan, Baoqi Liu, Yang Yu, and Tihua Yan. 2019. Computation offloading for mobile-edge computing with multi-user. In *Proceedings of the ICDCS*. IEEE, 841–850.
[12] Peter Friess. 2016. *Digitising the Industry—Internet of Things Connecting the Physical, Digital and Virtual Worlds*. River Publishers.
[13] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Comput. Commun. Rev.* 45, 5 (2015), 37–42.
[14] P. Georgiev, N. D. Lane, K. Rachuri, and C. Mascolo. 2016. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the ACM MobiCom*.
[15] Morteza Golkarifard, Ji Yang, Zhanpeng Huang, Ali Movaghar, and Pan Hui. 2018. Dandelion: A unified code offloading system for wearable computing. *IEEE Trans. Mob. Comput.* 18, 3 (2018), 546–559.
[16] Mark S. Gordon, Davoud Anoushe Jamshidi, Scott A. Mahlke, Zhuoqing Morley Mao, and Xu Chen. 2012. COMET: Code offload by migrating execution transparently. In *Proceedings of the USENIX OSDI*.
[17] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng. 2017. TinyLink: A holistic system for rapid development of IoT applications. In *Proceedings of the ACM MobiCom*.
[18] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. 2016. Operating systems for low-end devices in the internet of things: A survey. *IEEE Internet Things J.* 3, 5 (2016), 720–734.
[19] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky. 2006. PocketSphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Proceedings of the IEEE ICASSP*.
[20] Young Geun Kim, Young Seo Lee, and Sung Woo Chung. 2019. Signal strength-aware adaptive offloading with local image preprocessing for energy efficient mobile devices. *IEEE Trans. Comput.* 69, 1 (2019), 99–111.
[21] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. 2012. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the IEEE INFOCOM*.
[22] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of the ACM MobiCom*.
[23] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. 2015. Architecture-aware automatic computation offload for native applications. In *Proceedings of the ACM MICRO*.
[24] Jaemin Lee, Yuhun Jun, and Euiseong Seo. 2017. An enhanced DSM model for computation offloading. In *Proceedings of the IEEE PerCom*.
[25] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing public cloud providers. In *Proceedings of the ACM SIGCOMM*.
[26] Y. Li, Y. Chen, T. Lan, and G. Venkataramani. 2017. MobiQoR: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *Proceedings of the ICDCS*.
[27] P. Liu, D. Willis, and S. Banerjee. 2016. ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Proceedings of the SEC*.
[28] Nshmyrev. 2016. Language model and dictionary file of PocketSphinx. Retrieved from https://github.com/cmusphinx/pocketsphinx/tree/master/model/en-us.
[29] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.
[30] W. Wang, P. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba. 2017. Enabling cross-ISA offloading for COTS binaries. In *Proceedings of the ACM MobiSys*.
[31] Jian Yang, David Zhang, Alejandro F. Frangi, and Jing-yu Yang. 2004. Two-dimensional PCA: A new approach to appearance-based face representation and recognition. *Trans. Pattern Anal. Mach. Intell.* 26, 1 (2004), 131–137.
[32] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the ACM SenSys*.

Received March 2020; revised September 2020; accepted December 2020