Queec: QoE-aware Edge Computing for Complex IoT Event Processing under Dynamic Workloads

Gaoyang Guan[1], Wei Dong[1], Jiadong Zhang[1], Yi Gao[1], Tao Gu[2], Jiajun Bu[1]
{guangy,dongw,zhangjd,gaoy}@emnets.org,tao.gu@rmit.edu.au,bjj@zju.edu.cn
[1]College of Computer Science, Zhejiang University, and
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
[2]School of Science, RMIT University, Australia

ABSTRACT

Many IoT applications have the requirements of conducting complex IoT events processing (e.g., speech recognition) which 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 paper, 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 relative 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 art under dynamic workloads, while incurring acceptable overhead.

KEYWORDS

Internet of Things, Edge Computing, Offloading

ACM Reference Format:
Gaoyang Guan, Wei Dong, Jiadong Zhang, Yi Gao, Tao Gu, Jiajun Bu. 2019. Queec: QoE-aware Edge Computing for Complex IoT Event Processing under Dynamic Workloads. In *ACM Turing Celebration Conference - China (ACM TURC 2019) (ACM TURC 2019), May 17–19, 2019, Chengdu, China.* ACM, 5 pages. https://doi.org/10.1145/3321408.3321591

1 INTRODUCTION

Edge computing [3, 14], a new paradigm to complement cloud computing, is gaining 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 [12, 15] 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, etc.) 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) [10]. On the other hand, they are still the majority of commercial off-the-shelf (COTS) IoT devices and will dominate the IoT market for a few years [7]. Even the state-of-the-art edge computing platforms for IoT applications such as ParaDrop [13] 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 [10].

It is crucial to enable edge computing, especially task offloading, in low-end IoT devices to develop IoT applications in order to reduce the cost and facilitate large-scale deployment. In this paper, we propose a novel edge computing system which enables transparently offloading computation-intensive tasks from low-end IoT devices to nearby edge nodes or the cloud at runtime. However, turning the above ideas into reality faces two practical challenges.

First, it is difficult to design a common edge computing system which supports task offloading and is suitable for all low-end and high-end IoT devices, edge nodes and the cloud. Existing offloading approaches [5, 11–13] hardly address low-end IoT devices because they are too resource constrained to support these approaches, even many of them cannot support traditional OS like Linux or BSD [10].

Second, the state-of-the-art QoE-aware mobile edge computing framework, MobiQoR [12], proposes a novel framework to improve service response time by relaxing the QoE (e.g., service accuracy). However, its scheduler overlooks workloads at edge nodes which may lead to overloading that causes high offloading latency, as well as the multi-threaded execution ability of multi-core processors which may greatly reduce the offloading latency.

To address these problems, we present Queec, a QoE-aware edge computing system for complex IoT event processing under dynamic workloads. We carefully design Queec's system architecture and the workflow in order to reduce the overhead of the low-end IoT devices and the edge nodes, especially the gateway. Based on the RPC-based offloading, Queec can build lightweight IoT applications for low-end IoT devices and allow edge nodes to download the required offloading libraries on-demand at runtime. Combined with the TinyLink [9], Queec facilitates the rapid development of IoT applications with edge computing.

Queec presents a novel QoE-aware multi-user scheduling algorithm to minimize the overall offloading latency of all tasks. Queec avoids overloading by gathering the dynamic workloads

ACM TURC 2019, May 17–19, 2019, Chengdu, China

ACM ISBN 978-1-4503-7158-2/19/05…$15.00
https://doi.org/10.1145/3321408.3321591

```
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(
      file, result, hmm_model, language_model,
      dictionary); //speech to text recognition
4   if (!rtn && result.find("turn on the light")
      != string::npos) {
      ... //corresponding controlling commands
5   }
6 }
```

Figure 1: Example of application code skeleton.

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.

Section 2 introduces Queec via a use case study. Section 3 describes Queec's design and architecture. Section 4 evaluates the performance and the overhead. Section 5 discusses the related work, and finally, Section 6 concludes the paper.

2 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.

2.1 Sample Application

Since speech recognition and face recognition are widely-used in IoT community, we design two recognition applications. Voice controlled LED lamp in [9] is an application that automatically recognizes user's speech and controls the LED lamp correspondingly. It records the 16kHz audio as input and automatically offloads the recognition task to nearby edge nodes or the cloud. Face recognition application is to take a photo and recognize 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.

2.2 Development Process

We illustrate the process of application development with Queec by going through the speech recognition application. 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; and (4) connect the device to the WiFi AP. Developers need not to write any annotations about offloading. Figure 1 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 in order to register the device.

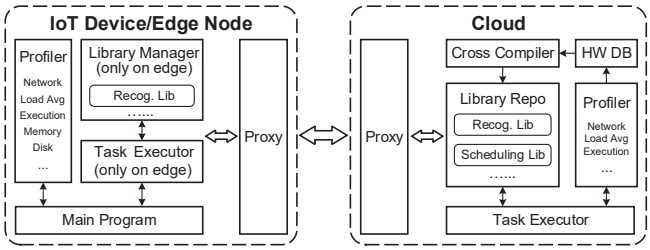


Figure 2: Queec architecture overview.

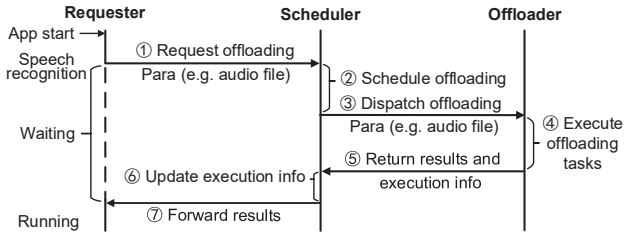


Figure 3: Workflow illustration.

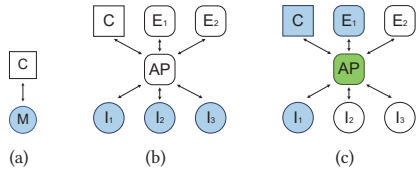


Figure 4: Topology of the offloading scenario.

3 QUEEC DESIGN

In this section, we present Queec's architecture and workflow. We then describe where to place the scheduler, as well as the formulation of the QoE-aware multi-tasks scheduling problem and how to optimize it.

3.1 Architecture and Workflow

Figure 2 shows Queec's overall architecture. Queec provides a light-weight homogeneous architecture for all edge nodes and IoT devices. The profiler monitors the statuses of the network (e.g., bandwidth, latency, etc.), the system (e.g., CPU workload, memory usage, etc.). 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 similar to the approach in [8]. Figure 3 shows the seven steps of a sample workflow which illustrates how the speech recognition API in Figure 1 is offloaded. Other detailed design considerations, e.g., how to quantify QoE of libraries, can be found in our technical report [2].

3.2 Scheduler Placement

Shifting existing offloading systems into IoT scenarios faces a challenge that is where to place the scheduler. Existing mobile systems [5, 6, 8] involve two kinds of participants, the Mobile device and the Cloud (M and C in Figure 4(a)). They usually run a scheduler at each mobile device. In IoT scenarios, two more participants exist, i.e., IoT devices and Edge nodes (I and E in Figure 4(b)). WiFi AP is the gateway edge node. Thus, it not only changes the types of participants, but also the network topology.

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, 4 cores	8 GB	50 GB HDD	1 Gbps Ethernet
	E_2	Dell OptiPlex 7050	Intel Core i7-6700 3.40 GHz, 4 cores	12 GB	100 GB HDD	1 Gbps Ethernet
Cloud	C	Aliyun ECS hfg5	Intel Xeon Gold 6149 3.10 GHz, 4 cores	16 GB	40 GB SSD	1 Gbps Ethernet

One solution is so-called *distributed scheduling* from [5, 6, 8, 12]. As shown in Figure 4(b), I_1, I_2 and I_3 perform their scheduling locally. However, two issues arise: (1) Each scheduler only generates its optimal solution locally which may lead to the overloading problem. On the other hand, synchronizing these decisions among devices may involve significant communication overhead. (2) It causes large redundancy since each scheduler 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 4(c), we have four candidates, i.e., C, I_1, 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 $T_{schd} = \max\{T_{req} + T_{schd\_exec}\}$.

We set up an experiment with the topology shown in Figure 4(c). Table 1 shows the hardware specifications. We let all IoT devices send an offloading request to the scheduler at the same time. To estimate T_{schd_exec}, we implement a linear programming problem, similar to [12]. Figure 5 shows the overall scheduling latency of each participant. We observe that E and AP achieve much lower overall scheduling latency than IoT devices and the cloud. We finally select AP as the centralized scheduler because it is the network center and the communication overhead can be greatly reduced.

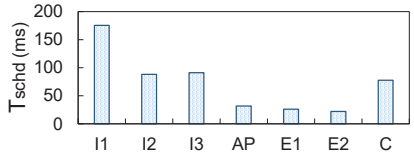


Figure 5: Overall scheduling latency of each participant as the centralized scheduler.

3.3 QoE-aware Scheduling

Once receiving offloading requests, the scheduler will search all offloaders to achieve the *minimal offloading latency* while maintaining required QoE and balancing tasks among offloaders.

3.3.1 Problem Formulation. We first introduce the 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 3, $M = \{I_1, I_2, I_3, E_1, E_2, AP, C\}$. N denotes the set of all tasks to be offloaded.
- i, j, d_{ij}: d_{ij} represents the indicator variable. $d_{ij} = 1$ if task j is designated to offloader i, where $i \in M,\ j \in N$. Otherwise, $d_{ij} = 0$ indicates 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}.
- l_{ij}, g, L_i^E, L_i^M: L_i^E represents the 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.
- 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.

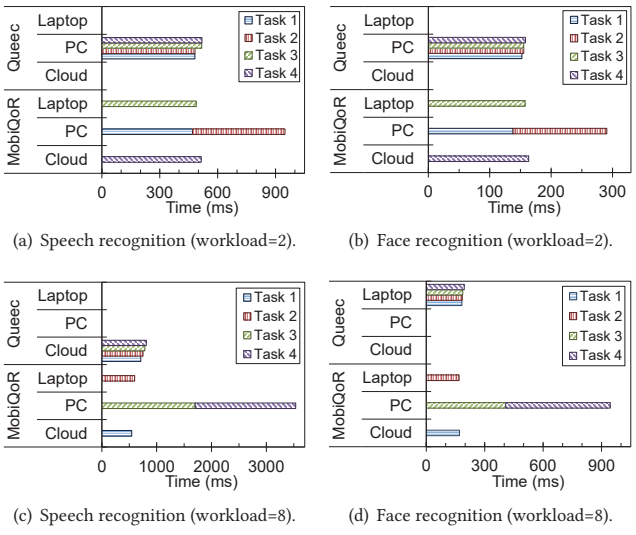
With the workflow in Figure 3, the minimal offloading latency for N offloading tasks can be written as:

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

This offloader selection problem can be formulated as an optimization problem with the 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 [4]. We add an auxiliary variable y to formulate the problem as follows:

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

We simplify the round-trip transmission time T_{ij}^O as $D_j/P_i + R_i$, because the task results are usually short plain text whose



(a) Speech recognition (workload=2). (b) Face recognition (workload=2).

(c) Speech recognition (workload=8). (d) Face recognition (workload=8).

Figure 6: Detailed offloading latency of MobiQoR and Queec under different workloads on the offloader, PC.

transmission time can be ignored. This problem is a mixed integer linear programming (MILP) problem which is NP hard. However, considering the scale of wireless local area network is not large, approximately 10∼100 computation nodes, it is acceptable to directly use general ILP solvers to resolve the problem. We describe how Queec estimates the execution time τ_{ij} and the load average l_{ij} by the trained models $f()$ and $g()$ with the input of QoE, workload and quantity in technical report [2].

3.3.2 Comparison with MobiQoR. MobiQoR [12] 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 twenty photos in MobiQoR's evaluation) into N pieces and distribute them to the edge nodes. We formulate MobiQoR's main scheduling algorithm as follows:

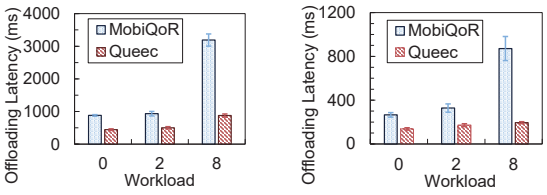
$$\begin{aligned} \min \quad & \max_{i \in M} [\sum_{j \in N} (\frac{D_j}{P_i} + R_i + \tau_{ij}) d_{ij}] \\ \text{s.t.} \quad & \sum_{i \in M} d_{ij} = 1, \quad \forall j \in N. \end{aligned} \tag{3}$$

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 workloads and can avoid overloading.

4 EVALUATION

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

Experiment Setup. We implement Queec on all computational nodes, as illustrated in Table 1. In addition, we also implement two low-end IoT devices using Arduino Due. The modules for IoT devices include WiFi Shield, Waveshare Music Shield, OpenJumper Camera module, USB Webcams and USB Microphones. 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 4(c).



(a) Speech recognition. (b) Face recognition.

Figure 7: Overall offloading latency of two recognition applications under different workloads on the offloader, PC.

We use the two sample applications illustrated in Section 2 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 [12], 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].

Offloading latency. We first have a detailed look at the offloading latency of both MobiQoR and Queec under different workloads. We use four low-end IoT devices, two LinkIt Smart and two Arduino DUE, to send one offloading request simultaneously. We set the powerful offloader, PC, to few workloads (i.e., workload=2) and full workloads (i.e., workload=8) before scheduling by running computation-heavy processes, respectively.

Figure 6 shows the detailed result. It is easy to observe that Queec reduces at least 45.33% of the overall offloading latency than MobiQoR in four 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 Figure 6(c) and (d), when the powerful offloader is overloaded, MobiQoR's scheduler still delivers two tasks to it, which causes dramatic increase in the offloading latency. On the contrary, Queec avoids overloading and fully utilizes offloaders' multi-core processors. Note that in Figure 6(c), Queec delivers speech recognition tasks to the far-away cloud because transmission causes less latency than the task execution.

Then we evaluate the overall offloading latency of two recognition applications under different workloads. This time, we use four low-end IoT devices to continuously send one offloading request to the scheduler simultaneously every second. Figure 7(a) and (b) show the results. We observe that Queec reduces 56.13% and 57.83% of the overall scheduling latency for the speech recognition and the face recognition, respectively. The more workloads on powerful offloaders, the more overall latency it introduces by MobiQoR's scheduling. The main cause is still that MobiQoR overlooks offloaders' workloads and multi-thread execution abilities.

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 3. In addition, we separate the execution of offloading tasks into the task execution and the preparation for execution which includes locating and dynamically loading the library. Figure 8 shows the

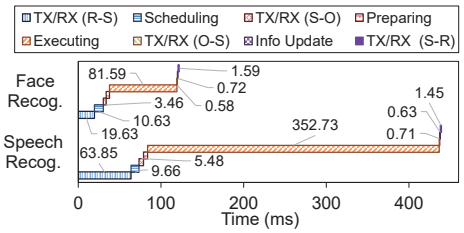


Figure 8: Breakdown of response time.

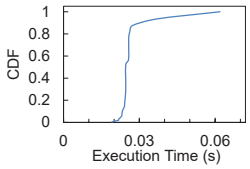


Figure 9: CDF curve of the scheduling time.

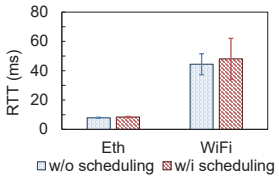


Figure 10: Scheduling influence on latency.

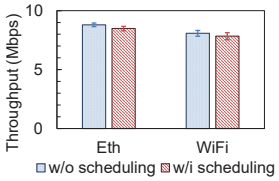


Figure 11: Scheduling influence on throughput.

result. We can observe that: (1) For both applications, the execution time is the main source of the response time, about 80.28% and 66.78%, respectively; (2) Excluding the transmission time and the execution time, Queec's basic procedure involves small latency, about 15 ms on average. (3) The transmission time of the request to the scheduler is usually much larger than the returned result, because the request usually contains large data, e.g., audio files, while the result contains plain text. Also the return does not perform a TCP three-way handshake as the connection is kept alive.

Overhead. We first evaluate the executing time of scheduling algorithm. Figure 9 shows that around 80% of the execution time are below 26 ms and none of them exceeds 63 ms, which is relative small. Then, we evaluate the scheduling influence in terms of latency and throughput. Four users are connecting to the AP via their smartphones and using live chat applications. We record RTT and throughput of the wired connection between the PC and the cloud, and the wireless connection between the smartphone and the cloud. We observe that in Figure 10 the latency increases slightly, i.e., 0.46 ms and 3.67 ms on average for the wired and the wireless, respectively. In Figure 11, the throughput drops a little, i.e., 3.59% and 2.95% on average for the wired and the wireless, respectively.

5 RELATED WORK

In this section, we discuss the related work from the two aspects.

Mobile computation offloading. A significant amount of previous work [5, 6, 11, 15] has explored the area of offloading computation intensive tasks from the resource-constrained mobile devices to the cloud so as to accelerate the task execution or reduce the battery drain. Some uses virtual machines to enable execution of offloading tasks from ARM on x86 servers, e.g., Android DVM [5, 11] and Microsoft .NET Framework [6]. Others use emulators to dynamically translate tasks with the heterogeneous binary to offloader's native binary [15]. However, the above approaches are not applicable in IoT scenarios because: (1) To the best of our knowledge, there is no unified VMs that can run on most of the existing IoT devices, especially low-end IoT devices. (2) Each architecture pair (e.g., ARM and x86), demands for a dedicated emulator, which is costly and impractical due to the severe heterogeneity of IoT devices. Unlike them, Queec adopts a practical RPC-based approach and implements a complete offloading system for low-end IoT devices and edge nodes.

Edge computing. Recent years have witnessed the innovations in a new computing paradigm, the edge computing [14] (a.k.a. fog computing [3]). Liu *et al.* propose and implements a WiFi AP based edge computing platform, ParaDrop [13], which supports multi-tenancy of WiFi APs and orchestrate APs through a cloud-based backend. Li *et al.* propose a mobile edge computation framework, MobiQoR [12], 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 them, Queec is a complete offloading system that can transparently offload tasks from IoT devices. Queec's scheduler takes more information about the offloaders' status (e.g., workloads) and capability (e.g., multi-core) as input and can achieve much better scheduling performance.

6 CONCLUSION

In this paper 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

This work is supported by the National Science Foundation of China (No. 61772465 and No. 61872437), Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars (No. LR19F020001) and the Fundamental Research Funds for the Central Universities (No. 2018FZA5013).

REFERENCES

[1] 2012. The AT&T Database of Faces. https://bit.ly/1xBJWDl.
[2] 2019. Queec Technical Report. http://ggypl.com/queec.pdf.
[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog computing and its role in the internet of things. In *Proc. of ACM MCC workshop on Mobile cloud computing*.
[4] C. Charalambous and A. Conn. 1978. An efficient method to solve the minimax problem directly. *SIAM J. Numer. Anal.* 15, 1 (1978), 162–187.
[5] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proc. of ACM EuroSys*.
[6] 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 *Proc. of ACM MobiSys*.
[7] Peter Friess. 2016. *Digitising the industry-internet of things connecting the physical, digital and virtual worlds.* River Publishers. 33–35 pages.
[8] P. Georgiev, N. D Lane, K. Rachuri, and C. Mascolo. 2016. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proc. of ACM MobiCom*.
[9] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng. 2017. TinyLink: A Holistic System for Rapid Development of IoT Applications. In *Proc. of ACM MobiCom*.
[10] 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 of Things Journal* 3, 5 (2016), 720–734.
[11] 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 *Proc. of IEEE INFOCOM*.
[12] Y. Li, Y. Chen, T. Lan, and G. Venkataramani. 2017. MobiQoR: Pushing the Envelope of Mobile Edge Computing via Quality-of-Result Optimization. In *Proc. of ICDCS*.
[13] P. Liu, D. Willis, and S. Banerjee. 2016. ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge. In *Proc. of IEEE/ACM Symposium on Edge Computing (SEC)*.
[14] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.
[15] W. Wang, P. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proc. of ACM MobiSys*.