Echo: An Edge-Centric Code Offloading System With Quality of Service Guarantee

LI LIN\(^{1,2}\) (Member, IEEE), PENG LI\(^3\) (Member, IEEE), XIAOFEI LIAO\(^{1}\) (Member, IEEE), HAI JIN\(^1\) (Fellow, IEEE), AND YU ZHANG\(^{1}\) (Member, IEEE)

\(^{1}\)Service Computing Technology and System Lab, Cluster and Grid Computing Lab, and School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
\(^{2}\)College of Mathematics and Informatics, Fujian Normal University, Fuzhou 350117, China
\(^{3}\)School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu 965-8580, Japan

Corresponding author: Xiaofei Liao (xfliao@hust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61502103 and Grant 61825202 and in part by the Natural Science Foundation of Fujian Province under Grant 2017J01737.

Abstract Code offloading is a promising way to accelerate mobile applications and reduce the energy consumption of mobile devices by shifting some computation to the cloud. However, existing code offloading systems suffer from a long communication delay between mobile devices and the cloud. To address this challenge, in this paper, we consider to deploy edge nodes in close proximity to mobile devices and study how they benefit code offloading. We design an edge-centric code offloading system, called Echo, over a three-layer computing hierarchy consisting of mobile devices, the edge, and the cloud. A critical problem needs to be addressed by Echo is to decide which methods should be offloaded to which computing platform (the edge or the cloud). Different from existing offloading systems that let mobile devices individually make offloading decisions, Echo implements a centralized decision engine at the edge. This edge-centric design can fully exploit limited hardware resources at the edge to provide offloading services with the quality-of-service guarantee. Furthermore, we propose some novel mechanisms, e.g., lazy object transmission and differential object update, to further improve system performance. The results of a small-scale real deployment and trace-driven simulations show that Echo significantly outperforms existing code offloading systems at both execution time and energy consumption.

Index Terms Code offloading, edge computing, offloading decision, quality of service.

I. INTRODUCTION

Mobile devices have evolved significantly with faster CPU and larger memory in recent years. However, they are still constrained to run large applications, such as augmented reality and games, due to limited hardware resources and battery capacity. Code offloading [1]–[3] has been proposed to conquer this challenge by shifting some computation tasks of mobile devices to the cloud. It can potentially save battery energy of mobile devices and accelerate mobile applications by using powerful hardware in the cloud. However, the cloud is usually located geographically far from mobile devices, leading to limited network bandwidth and long transmission latency, which cannot meet the latency requirements of real-time applications.

Edge computing [4], [5] emerges as a promising paradigm that deploys a number of modest-size computing nodes in the proximity of mobile devices, so that the computing requests from mobile devices can be quickly served with low latency. Many research efforts have been made to exploit the benefits of edge computing, such as the deep neural networks over end devices, the edge, and the cloud [6], multi-level IoT systems [7], vehicular computing [8], online algorithms for service reconfiguration at the edge [9], and the deployment of edge nodes [10]. However, the integration of code offloading and edge computing with the performance guarantee is still an open challenge.

The comparison of typical code offloading systems is shown in Table 1. Most of existing code offloading systems [1]–[3], [11] are designed for a two-layer computing hierarchy (i.e., mobile devices and the cloud), unaware of the existence of edge nodes. Specially, the terminologies “edge” and “edge nodes” used in this paper refer to the...
TABLE 1. Comparison of code offloading systems.

| Offloading systems | Edge supported | Who make offloading decisions | Easy to use | Granularity | QoS guarantee |
|--------------------|----------------|------------------------------|-------------|-------------|---------------|
| MAUI [1]           | No             | Mobile Devices               | No          | Thread-level| No            |
| CloneCloud [2]     | No             | Mobile Devices               | No          | Thread-level| No            |
| COMET [11]         | No             | Mobile Devices               | Yes         | Method-level| No            |
| mCloud [12]        | Yes            | Mobile Devices               | No          | Method-level| No            |
| Echo               | Yes            | Edge Nodes                   | Yes         | Method-level| Yes           |

“cloudlets” [13], powerful machines deployed at the edge of the network. Extending these systems to support edge computing is not an easy task, which involves the redesign of the whole framework as well as the algorithm for offloading decisions. The first major challenge of building a three-layer, mobile-edge-cloud, code offloading system is to combine the three heterogeneous computing platforms and decide on which platform to perform the computation offloading for the best performance improvement. However, the traditional way lets mobile devices individually decide whether a portion of code should be offloaded according to application specifications, the quality of network connection [14], etc., as shown in Fig. 1(a). It is a reasonable design for the traditional two-layer computing hierarchy because the cloud has sufficient resources to quickly serve all computing tasks. Since there is little or no contention among tasks in the cloud, it is easy for each mobile device to estimate the expected task completion time in the cloud and choose to offload code if it is faster than its local execution. Unfortunately, edge nodes have limited resources, and tasks need to contend for running. Without the knowledge of tasks from other devices, it is difficult for a mobile device to estimate the task completion time at the edge and make right offloading decisions.

In this paper, we propose an edge-centric code offloading system, called Echo, over a three-layer computing hierarchy consisting of mobile devices, the edge, and the cloud. Echo implements a method-level code offloading on Android-based devices. It allows programmers to use an annotation, i.e., @Offloadable, to annotate methods that are considered to be offloaded. When an annotated method is invoked, the mobile device sends an offloading request to the edge, which then makes centralized offloading decisions, as shown in Fig. 1(b). To optimize the resource usage at the edge, we propose a novel task scheduling algorithm, called Preemption-Constrained Shortest-Remaining-Time-First (PC-SRTF), which aims to minimize the average task completion time without any prior knowledge of future task arrivals. Its basic idea is to let tasks with less remaining time preempt current running tasks, only if the running tasks can finish no later than their local and cloud execution. The decision engine estimates the expected task completion time at the edge according to PC-SRTF, compares it with the completion time on the local and the cloud, and assigns the task to the fastest platform. If a task is decided to be offloaded to the edge, Echo can provide Quality of Service (QoS) guarantee, i.e., even though a task is preempted by future tasks, it can still complete no later than running on mobile devices and in the cloud. For offloaded methods, we further optimize their data uploading with two mechanisms, lazy object transmission and differential object update, to reduce the amount of data transmission over the network, so as to reduce the end-to-end delay.

The proposed edge-centric design has three main advantages. First, centralized decision making at the edge can provide improved and predictable performance. Second, edge nodes can interact with mobile devices and the cloud in an asynchronous way, leading to reduced overhead. For example, when a method is offloaded to the edge, edge nodes synchronize the runtime environment with mobile devices. After that, edge nodes upload the same runtime environment to the cloud, so that future offloaded methods running in the cloud can use it. This process does not block the interaction between edge nodes and mobile devices. More details can be found in Section V. Finally, Echo can easily deploy new offloading policies by updating the centralized decision engine at the edge, without changing the applications on mobile devices. The main contributions of this paper are summarized as follows.

1) We design and implement a distributed computing framework over mobile devices, the edge, and the cloud for code offloading.

2) At the edge, we design a centralized decision making algorithm based on PC-SRTF. It can optimize the resource usage at the edge, with guaranteed offloading performance for mobile devices.
3) We enhance Echo’s performance by some novel designs, e.g., data transmission optimizations, and quick service provisioning at the edge.

4) A small-scale real deployment and trace-driven simulations are conducted to evaluate the performance of Echo. The results show that Echo outperforms existing code offloading systems at both the average task completion time and energy consumption of mobile devices.

The rest of this paper is organized as follows. Some important related works are reviewed in Section II. System overview is presented in Section III. Section IV introduces the centralized decision engine at the edge, followed by other design details in Section V. Section VI presents system implementation and Section VII shows experimental results. Finally, Section VIII concludes this paper.

II. RELATED WORK

Code offloading has attracted many research efforts in recent years [1]–[3], [11], [15]. Chun et al. [2] have proposed CloneCloud, which can automatically migrate code execution associated with the clone of mobile device’s OS to the cloud at thread level. Based on CloneCloud, COMET [11] has implemented code offloading built on distributed shared memory to support multi-threaded applications. However, the implementation of code offloading at the thread level needs the customization of mobile runtime systems, which is intricate and difficult for large-scale deployment. MAUI [1] has designed a code offloading framework at the method level in order to reduce the energy consumption of mobile devices. ThinkAir [3] has further optimized resource allocation in the cloud, so that it can provide better offloading services to multiple mobile users. Although both MAUI and ThinkAir adopt annotations to implement the method-level offloading, which is similar to Echo, they are not designed to support the paradigm of edge computing. Furthermore, they do not study offloading decisions to guarantee quality-of-services for multiple users, which is one of the main contributions of Echo.

Edge computing [4], [5] has attracted great interests from both industry and academic [16]. As the network proximity, edge nodes can achieve lower response delay with mobile devices than the cloud [17]. Tong et al. [10] have designed a hierarchical edge-cloud architecture to handle offloading requests from mobile devices and further proposed a workload placement algorithm to maximize utilization of cloud resources. Hou et al. [9] have proposed an online algorithm to configure edge-clouds based on history knowledge, which aims to improve the performance of mobile edge computing and minimize the cost. Chen et al. [18] have studied the problem of multi-user code offloading in mobile edge computing and proposed a distributed offloading decision algorithm based on game theory. Jia et al. [19] have proposed an algorithm to properly place edge nodes across wireless metropolitan area network. Above works mainly focus on algorithm design and theoretical analysis for general workloads, but lack of implementation and optimization of practical code offloading systems.

Satyanarayanan et al. [13] have first introduced the concept of “cloudlet”, a typical paradigm of edge nodes. A cloudlet, having good network connection with mobile devices, resembles a small data center deployed at airport, coffee shop, hospital, etc. Specially, The terminologies “edge” and “edge nodes” used in this paper refer to cloudlets. Ha et al. [20] have built Gabriel for wearable cognitive assistance based on cloudlets. Later, they built OpenStack++ [21] to provide system infrastructure supports. Wu et al. have also built infrastructures for code offloading based on containers and unikernels [22]–[24]. Wang et al. [25] have implemented a system for privacy-aware live video analytics on cloudlets. Glimpse [26] is a real-time object recognition system with cloudlet assistance. The above systems focus on specific applications and need great efforts to customize the runtime and the infrastructure of edge computing. In contrast, Echo is a versatile code offloading solution that can support quick deployment of a wide range of applications. Similar to Echo, mCloud [12] builds a context-aware offloading framework over mobile devices, cloudlets, and the cloud, but it lets mobile devices individually make offloading decisions, leading to unpredictable offloading performance.

III. OVERVIEW OF ECHO

A. PROBLEM STATEMENT

The main goal of Echo is to build a code offloading framework over mobile devices, the edge, and the cloud. It implements a method-level offloading on Android-based devices, without any modifications to mobile operating systems and runtime environments. A critical challenge needs to be addressed by Echo is to decide which methods should be offloaded to which computing platform (the edge or the cloud), so as to accelerate mobile applications.

A typical scenario of using Echo is as follows. Suppose some companies use Echo to develop and deploy mobile applications with offloading capability. Echo provides an annotation-style API, i.e., @Offloadable, and programmers use it to annotate methods that are considered to be offloaded, as shown in Fig. 2. This annotation requires little knowledge on method details, and it works as only a suggestion to Echo’s decision engine, which will make final offloading decisions during execution. After annotation, applications are released for downloading and installation. Meanwhile, companies start to provide offloading services by deploying the Echo system as well as the application code at the edge and in the cloud. They can use public cloud services or deploy their own...
estimates the expected task completion time at the edge and makes a final offloading decision that is sent back to the mobile device. Specifically, it can provide QoS guarantee when computation offloaded to the edge.

If an method is decided to be offloaded to the edge, Echo quickly prepares an executor based on a virtual machine (VM) with configured runtime at the edge and synchronizes its state with the one at the mobile device. This process is called VM provisioning. Meanwhile, with the method execution, we also need to upload input data and objects involved in this method. To optimize this synchronization, we propose a mechanism called lazy object transmission, which synchronizes an object only when it is used during execution. Furthermore, Echo caches transmitted objects at the mobile and the edge and adopts the differential object update. That is, when the same object is synchronized again, Echo compares it with the cached one and transmits only the different parts. The above techniques are also applied on executors in the cloud.

IV. CENTRALIZED DECISION ENGINE AT THE EDGE
The decision engine at the edge is the most critical module of Echo to provide offloading QoS guarantee. We design the decision engine with the objective to minimize the average task completion time, without prior information of future offloading requests. We first present the system model and then describe the decision-making algorithm based on PC-SRTF.

A. SYSTEM MODEL
We consider an edge node with limited hardware resources, which are virtualized as a set $M$ of identical virtual machines (VMs). As mentioned above, tasks are executed based on virtual machines, and a VM only serves one task at a time in Echo, which is a common way in existing works [10], [27]. As a result, an edge node has limited virtual machines can only serve a given number of tasks at a time. Offloading tasks generated by mobile devices arrive in an online manner, i.e., the edge has no knowledge about future task arrivals. For each task $i$, its completion time $T(i)$ is defined as follows.

$$T(i) = \begin{cases} R_m(i), & \text{mobile device;} \\ D_e^{up}(i) + W_e(i) + R_e(i) + D_e^{down}(i), & \text{edge;} \end{cases}$$  

(1)

Case 1: task $i$ runs at the mobile device. The task completion time $T(i)$ is equal to the local running time $R_m(i)$, which is determined by the task size and the hardware of the mobile device, and it can be predicted by the profilers illustrated in Section V-B.

Case 2: task $i$ is offloaded to the cloud. The mobile device needs to upload data to the cloud, and the uploading time is denoted by $D_e^{up}(i)$. We suppose the cloud has sufficient resources, and the task can be immediately served by $R_e(i)$ time. After execution, the mobile device downloads results and the downloading time is denoted by $D_e^{down}(i)$.
Case 3: task \( i \) is offloaded to the edge. The data uploading and downloading time are denoted by \( D_{\text{up}}^i \) and \( D_{\text{down}}^i \), respectively. Different from the cloud, an edge node has limited resources, i.e., the limited virtual machines in a single node. Then, the task may need to wait before being served due to resource contention. Therefore, the time spent at the edge consists of the task waiting time \( W_e(i) \) and the running time \( R_e(i) \). The task waiting time \( W_e(i) \) depends on the task scheduling algorithm adopted by the edge.

The profiling modules, deployed at mobile devices, the edge, and the cloud, collect the information of task size, VM power of different computing platforms and quality of network connection (details in Section V-B), so that the decision engine can accurately estimate the task running time \( (R_m(i), R_e(i), \text{and } R_c(i)) \), data uploading time \( (D_{\text{up}}^i \text{ and } D_{\text{down}}^i) \) and downloading time \( (D_{\text{down}}^i \text{ and } D_{\text{down}}^i) \). Since edge nodes are located closer to mobile devices than the cloud, we have \( D_{\text{up}}^i < D_{\text{up}}^i \) and \( D_{\text{down}}^i < D_{\text{down}}^i \). Moreover, we usually have \( R_e(i) \leq R_e(i) \leq R_m(i) \) because the edge node and the cloud have more powerful hardware than mobile devices. Note that tasks have different characteristics on data transmission and task execution. For example, some tasks with little computation need to upload a large amount of data, whereas others (e.g., compute-intensive tasks) upload small data, but the execution is time-consuming.

### B. DECISION MAKING BASED ON PC-SRTF

For each offloading request \( i \), the decision engine estimates the expected task completion time at mobile devices, the edge, and the cloud, respectively, according to (1). It then makes an offloading decision by assigning the task to the platform with the minimum completion time.

We let \( T_m(i) \) and \( T_c(i) \) denote the completion time of task \( i \) on the mobile device and in the cloud, respectively. Based on measurements by the profilers, we can easily calculate \( T_m(i) \) and \( T_c(i) \). Next, we focus on the task scheduling at the edge, so that we can calculate the expected task completion time at the edge. Given a number of tasks offloaded to the edge, to minimize the average task completion time, an intuitive design is to always schedule the task with the minimum remaining time. Unfortunately, this simple heuristic would lead to starvation for long tasks if small tasks frequently arrive. When a task is offloaded to the edge, its completion time should be no later than a deadline \( H(j) = \min[T_m(i), T_c(i)] \). Otherwise, this task should be assigned to the cloud or the mobile device. This observation motivates us to design an algorithm called Preemption-Constrained Shortest-Remaining-Time-First (PC-SRTF), which allows smaller tasks to preempt the execution of longer tasks, only if the preemption makes longer tasks complete no later than their deadlines.

The pseudo code of the decision-making algorithm based on PC-SRTF is shown in Algorithm 1. For each VM \( m_q \in M \), we maintain a queue \( Q_q \) for waiting tasks. The VM \( m_q \) sequentially executes the tasks in \( Q_q \). When a new task \( i \) arrives at time \( t \), we calculate its expected completion time at the edge from line 2 to 24. Since task \( i \) is not really scheduled during this stage, we define a temporary queue \( Q_q' = Q_q \) in line 4 and do the following calculation based on it.

We search queue \( Q_q' \) to find the first task \( j \) whose remaining time is greater than task \( i \). If such a kind of task \( j \) cannot be found, we put task \( i \) at the tail of \( Q_q' \). Otherwise, a queue \( Q_q^{\text{ins}} \) is defined to include the tasks that will be inserted into \( Q_q' \), and it is initialized as \( (i) \). We let \( p \) point to the header of task \( j \), where we will insert \( Q_q^{\text{ins}} \). In the following while loop from line 11 to 20, we insert \( Q_q^{\text{ins}} \) at the place pointed by \( p \). If any task \( k \) completes later than its deadline \( H(k) \) due to this insertion, we need to adjust the scheduling by postponing \( k \) until it can finish at \( H(k) \). There must be

---

**Algorithm 1** Decision Making Algorithm Based on PC-SRTF

1. Maintain a task queue \( Q_q \) for each VM \( m_q \in M \);
2. for each new task \( i \) arriving at time \( t \) do
3.     for each VM \( m_q \) do
4.         \( Q_q' = Q_q \);
5.         Search queue \( Q_q' \) from the beginning, and find the first task \( j \) whose remaining time is greater than task \( i \);
6.         if cannot find such a task \( j \) then
7.             Put task \( i \) at the tail of \( Q_q' \);
8.         else
9.             \( Q_q^{\text{ins}} = (i) \);
10.            \( p \) → the header of task \( j \);
11.           while \( p \) is not NULL do
12.               Insert \( Q_q^{\text{ins}} \) at \( p \);
13.               if any task \( k \) completes later than \( H(k) \) then
14.                   Postpone task \( k \) until it can finish at \( H(k) \);
15.                   A portion of workloads before \( k \) are evicted, and use them to replace the contents in \( Q_q^{\text{ins}} \);
16.               \( p \) → the tail of task \( k \);
17.           else
18.               \( p \) → NULL;
19.           end if
20.         end while
21.     end if
22.     Calculate the completion time growth \( \Delta T_q \) of tasks in \( Q_q' \);
23. end for
24. \( q^* = \arg \min[\Delta T_q] \);
25. Compare \( T_m^q(i), T_c(i), \text{and } T_m(i) \), and assign the task \( i \) to the platform with the minimum completion time;
26. if task \( i \) should be offloaded to the edge then
27.     Assign task \( i \) to \( m_q \) and replace queue \( Q_q' \) with \( Q_q^{\text{ins}} \);
28.     \( H(i) = t + \min[T_m(i), T_c(i)] \);
29. end if
30. end for
in the bracket indicates the task size. Then, task $A_4$ with the size of 5 arrives at the 3rd time slot, but it cannot preempt any running tasks because their remaining time is less than $A_4$. We choose to schedule $A_4$ after $A_1$ on $m_1$ because it leads to the minimum growth of average completion time. Later, task $A_5$ and $A_6$ arrive, and it is easy to see that they should be scheduled on $m_2$ and $m_3$, respectively. At the 7th time slot, task $A_7$ arrives. It cannot preempt $A_4$ on $m_1$ because $A_4$’s remaining time is not more than $A_7$. The completion time growth is 10 if $A_7$ is scheduled after $A_4$. On $m_2$, we suppose that $A_5$’s deadline is the 11th time slot, and $A_7$ can preempt $A_5$ only at the 7th time slot, leading to the growth of 13 in average completion time. On $m_3$, $A_7$ can run after $A_3$ and the growth is 11. Therefore, we finally decide to schedule $A_7$ after $A_4$ on $m_1$.

V. DESIGN DETAILS

Echo is designed as an integrated offloading system, and we develop functional modules to guarantee its QoS and improve the overall system performance. The details of these modules are presented in this section.

A. EDGE DISCOVERY

The first step of edge-assist offloading is edge discovery. According to the different types of network connections between mobile devices and edge nodes, we develop a two-level edge discovery mechanism: a local discovery and a global one. The edge discovery modules, deployed on the mobile device, the edge, and the cloud, work collaboratively to perform the two-level discovery.

The local edge discovery uses the zero-configuration protocol (e.g., Bonjour protocol)\(^1\) to find available edge nodes that are in the same network domain (i.e., LAN) with mobile devices. In the local discovery, an edge node announces its offloading service by network broadcast, which includes the message of service name, IP address and port number. An example of this broadcast is _echo.<10.136.3.71>._<8022>_. A mobile device searching for this type of service (e.g., the echo service) receives the broadcast and negotiates with the edge, then finally connects to the edge node with the specific port.

The global discovery uses the cloud to help the discovery. The cloud acts as a directory server, which holds items of all available edge nodes geographically dispersed. Edge nodes register to the cloud with the information of IP address, location (longitude and latitude) and resource availability. Then they continuously send heartbeat messages indicating their states (active or disconnected). A mobile device asking for offloading services sends to the cloud a request, which includes user id, application id, IP address and the location of the mobile device. To respond this query, the cloud searches its database and finds an edge node that is close to the mobile device according to their location proximity.

---

\(^1\)https://en.wikipedia.org/wiki/Bonjour_(software)
B. ECHO PROFILERS

Echo’s edge-centric decision is based on the estimation of application execution time on the mobile device, the edge, and the cloud, respectively. For a precise estimation, we use the way of history-based profiling prediction, which has been substantiated by previous works [1], [28]. To build the prediction model, we collect resource information and code behavior by profilers, which include hardware profiler, network profiler, and code profiler, deployed on mobile devices, the edge, and the cloud, shown in Fig. 3. The hardware profiler provides hardware state about the power of CPU, memory, etc. with respect to mobile devices, the edge, and the cloud. The network profiler probes network characteristics, such as network type, bandwidth, and network latency. The code profiler observes program behavior when the program runs with different input data, and it records the size of input data, the number of method instructions and the overall size of instructions, the resource requirement (e.g., CPU cycles, bandwidth, and latency). Then, these history logs feed the prediction model, which can estimate the execution time of an offloading method at runtime.

In more detail, we illustrate how to predict the $R_{m}(i)$, $R_{r}(i)$, $R_{e}(i)$, etc. in (1). First, for the network transmission time, including $D_{e}^{up}(i)$, $D_{e}^{down}(i)$, $D_{c}^{up}(i)$, and $D_{c}^{down}(i)$, it is easy to capture the transmission data with the network bandwidth and latency, and so to estimate the network transmission time. For every data transmission, we will recalculate the bandwidth and latency and always use the latest network parameters to have a good estimation. Then, for the prediction of the method execution time on mobiles, i.e., $R_{m}(i)$, we use the above profiling based technique. Specifically, we run each application with different inputs on different mobile phones and measure the program characteristics, such as the number of instructions, the resource requirement (e.g., CPU cycles and memory), and the running time. Then, we label these logs as training data for off-line training. To have a trade-off between the prediction error and accuracy, we employ the linear-fit predictor for our training data [28]. Then the trained prediction function can perform an online prediction about the $R_{m}(i)$, and it will update the coefficients of the function continuously [29]. Finally, the prediction of $R_{r}(i)$ and $R_{e}(i)$ uses the same way as the $R_{m}(i)$ does.

1) CODE PROFILER

As discussed above, the code profiler measures the running data of method execution. Besides, it provides a feature of code inspection on annotated methods, to check if those methods can be offloaded. We build call graphs [30] to analyze if the annotated methods contain unoffloadable operations, e.g., hardware control, UI manipulation, etc., and those methods containing these operations should be anchored on mobile devices. The flowchart of code analysis is shown in Fig. 6, and the analysis module is designed for Android program. In the beginning, it takes an APK file (an Android application zip file) as input and analyzes the AndroidManifest.xml file to get the main entrance of the whole application. The module then performs a rule-based dataflow analysis. In our current implementation, the following rules are used.

Rule 1. Mobile hardware operations must be anchored on mobile devices. These operations involve using hardware sensors like camera, GPS, microphone, etc.

Rule 2. UI manipulations must be anchored on mobile devices. These operations involve using hardware sensors like camera, GPS, microphone, etc.

Rule 3. IO (Input/Output) operations must be anchored on mobile devices. This rule filters the read and write operations of IO streaming including file IO and network IO. It also limits data storage operations like using SQLite database or SharedPreference.

Rule 4. Graphics rendering and display must be anchored on mobile devices. Android offers a variety of graphics rendering APIs, for example, OpenGL ES for Android. These APIs interact with mobile graphic drivers which are hardware-dependent.

C. DATA TRANSMISSION OPTIMIZATIONS

When Echo decides to offload a method, it synchronizes the state of the executor by uploading related objects from mobile devices to the edge or the cloud. We design two mechanisms to accelerate this synchronization.

1) LAZY OBJECT TRANSMISSION

In an intuitive design, all objects associated with an offloaded method should be transmitted to the edge or the cloud. However, by carefully examining the code execution, we find that not all these objects will be used. This observation motivates us to design a lazy object transmission mechanism [31]. Specifically, when a method is offloaded, we create proxies for associated objects at the edge or in the cloud, instead of transmitting real ones. Once an object is actually used, an object transmitting request is sent back to the mobile device, which then transmits the real one. As a result, we remove unnecessary object transmission.

2) http://en.wikipedia.org/wiki/SQLite
3) https://developer.android.com/training/data-storage/shared-preferences.html
4) https://developer.android.com/guide/topics/graphics/opengl
Virtual machine (VM) based code offloading can provide a consistent runtime for code execution across heterogeneous computing platforms. It has been widely adopted by existing works, e.g., [1] and [2], which start a dedicated VM for each mobile device. The dedicated VM is usually created by launching a base VM that then synthesizes a differential VM binary, which is the different part of VM between the base one and the mobile device’s specific VM, and this process is called VM synthesis [27]. This way is easy to be deployed but low resource utilization; moreover, mobile devices should transmit their VM binary files to the server, which is time-consuming. Conversely, in Echo, for quick offloading service provisioning, we propose a lightweight VM synthesis by reusing existing VMs and setting up the execution environment for different users. To provide an executable VM for a mobile device, we launch a base VM and install application packages associated with configurations for the specific mobile device. Furthermore, when a mobile device searches offloading services for a specific application at the edge, we first choose the VMs that have installed the application package with corresponding program’s data. Therefore, the edge node can quickly respond to the mobile device without heavy data transmission.

VI. IMPLEMENTATION

We implement Echo using Java for Android mobile devices. It uses the paradigm of aspect-oriented programming (AOP), which allows us to insert specific offloading operations into application source code. To integrate Echo with Android applications, we use aspectj, which can recompile Java source code to enable code offloading. We use JSON for object serialization because it is more versatile and efficient compared to Java native serialization. Offloaded methods are executed at the edge or in the cloud by using Java Reflection.

The execution environment at the edge is built on the virtual machine of Android-x86, which can run Android applications on x86 platforms.

The code analysis module of Echo is based on FlowDroid [30], which provides a dataflow analysis framework for Android. In our implementation, we build a dummy main entry and perform static code analysis based on our customized rules. For better graphical user interface (GUI) objects and user-driven callbacks analyses, we employ GATOR, which offers a more precise inter-component communication analysis. The above tools are based on Soot [34], which is a framework for analyzing and transforming Java and Android applications.

VII. EVALUATION

In this section, we first introduce our experimental methodology and then present the results of a small-scale real deployment and trace-driven simulations.

A. METHODOLOGY

We build a small-scale testbed in our laboratory building to evaluate the performance of Echo. As shown in Fig. 8, 5

![FIGURE 7. An example of lazy object transmission. Initially, only object proxies are sent to the remote server. Once an object is actually used at runtime, we then retransmit the real object, e.g., the object a.](image-url)
we set up an edge node equipped with i7-4720HQ (2.60GHz with 4 cores of 8 threads) and 16GB RAM. The edge node is installed with Windows 8.1, and it launches Android-x86 VMs using VMware Workstation. Android-x86 is a project that ports Android OS to x86 platforms. Each Android-x86 VM is assigned 2 virtual cores and 4GB memory. We organize 10 people to run mobile applications in the experiments, and each one has a Samsung Galaxy N7000 smartphone with a dual-core 1.4GHz CPU and Android OS 4.4. Smartphones connect to the edge node via WiFi network, which achieves a better offloading performance compared to 3G and 4G network [35], [36]. We rent VM instances with 2-core CPU and 4GB memory in the AlibabaCloud. The network bandwidth of each VM instance in the AlibabaCloud is 1Mbps.

Mobile users run five open-source applications as benchmarks, including three interactive applications and two classic compute-intensive ones, which are summarized in Table 2. We modify their source code by annotating complex computing methods to be offloaded. For example, in the OCR, we choose the method that performs the text extraction algorithm. We use PowerTutor [37] to measure the energy consumption of smartphones.

For comprehensive evaluations, we compare Echo with the following systems.

- **End-only**: all methods are executed on mobile devices.
- **Cloud-always**: all annotated methods are offloaded to the cloud, and it is implemented based on ThinkAir.
- **ThinkAir** [3]: only the cloud is available for code offloading. Each annotated method is offloaded to the cloud if offloading is faster than the local execution. Otherwise, this method is executed by the mobile device.
- **mCloud** [12]: an edge assisted offloading system whose offloading decisions are made by mobile devices individually (unlike Echo), and it offers a best-effort service at the edge without data transmission optimization. Echo and mCloud are both three-layer offloading systems, but Echo performs an edge-centric offloading decision with PC-SRTF.

### B. PERFORMANCE OF REAL DEPLOYMENT

#### 1) OVERALL PERFORMANCE

We first study the task completion time by normalizing the results of all systems to the performance of Echo. For each application, the results are the average of 10 test runs (one per person) under the five system designs. As shown in Fig. 9(a), Echo significantly outperforms other systems on interactive applications. Compared to cloud-always, Echo can reduce the average completion time by about 69.7% and 92.4% on the OCR and ImageFilter, respectively. The offloaded methods in the OCR and ImageFilter usually involve heavy data transmission but moderate computation. Therefore, in this case, Echo and mCloud show great advantages because of quick data uploading to the edge than to the cloud. Echo can further outperform mCloud, thanks to our proposed data transmission optimizations. On compute-intensive applications, i.e., Sudoku and N-Queens, Echo achieves similar performance with ThinkAir and cloud-always, but they significantly outperform end-only. It is a direct evidence that offloading can get obvious benefits to migrate complex computation. However, since these compute-intensive applications have little data for uploading, the benefits of the edge proximity to mobile devices are not obvious. Moreover, since there is little resource contention at edge VMs in small-scale experiments, methods performing compute-intensive tasks have similar execution time at the edge and in the cloud with similar VM configurations.
The normalized energy consumption of smartphone is shown in Fig. 9(b). On the OCR and ImageFilter, Echo, mCloud, and cloud-always save more energy than the other two systems because they migrate part of method execution to the remote; meanwhile, the energy consumption of data transmission to the edge is more efficient than to the cloud, as the value of Echo and mCloud is much smaller than cloud-always. Moreover, Echo is more energy-efficient than mCloud due to less data transmission for our data transmission optimizations. Table 3 provides more details by showing the energy consumption of CPU and network interface. On the OCR and ImageFilter, Echo saves 86.3% and 93.9% WiFi energy, respectively, compared to cloud-always. It is also obvious that computation offloading to the edge or the cloud can sharply reduce the energy consumption of mobile devices on compute-intensive applications.

We further study the execution time breakdown of methods offloaded to the edge and the cloud. As shown in Fig. 10, the left bar and the right bar in each group show the time breakdown between computation and network transmission at the edge and in the cloud respectively. The number on top of each bar shows the total execution time in seconds. Thanks to the proximity of edge node, smartphones can always quickly get the results by offloading methods to the edge. On interactive applications, a large portion of the time is spent on data transmission. This phenomenon is more obvious when methods are offloaded to the cloud. For example, 99.6% of the time is spent on data transmission on the ImageFilter. In contrast, the computing takes up most of the time on compute-intensive applications, as it spends more than 90% of the time on the computing for the N-Queens.

2) PERFORMANCE OF LAZY OBJECT TRANSMISSION
We use the Android-OCR application as a case study to evaluate the performance improvement by lazy object transmission (in Section V-C1). In the implementation of Android-OCR, it first searches the mobile storage to load the image set, which contains images to be selected for text extraction. When the method responsible for text extraction is offloaded, the original image set is transmitted to the server. However, the images selected by users can only be determined at runtime. In this case, the lazy object transmission can effectively avoid transmitting images that are not selected by users so as to remove unnecessary data transmission. Table 4 compares the performance of Android-OCR with and without the lazy object transmission. In this example, the original image set includes 6 pre-installed images. If the user chooses only one image for text extraction, we can significantly reduce the data transmission time and energy consumption.
TABLE 4. Comparison of the transmission data, time, and energy consumption with and without the lazy object transmission.

|                     | Without lazy object transmission | With lazy object transmission |
|---------------------|----------------------------------|-------------------------------|
| Transmission Data   | 1951.0 (KB)                     | 262.9 (KB)                    |
| Transmission Time   | 3.9 (S)                          | 3.5 (S)                       |
| Transmission Energy | 450 (mJ)                         | 116 (mJ)                      |

3) PERFORMANCE OF DIFFERENTIAL OBJECT UPDATE

In Echo, we use the differential object update (DOU, in Section V-C2) to reduce the amount of data transmission during code offloading. In the DOU, object data are cached at both mobile devices and the remote (the edge or the cloud), and only the differential part of object needs to be transmitted when an object updates. We measure the execution time of offloaded methods with and without DOU respectively and show the normalized results in Fig. 11. We observe that method execution with DOU can significantly reduce the method execution time, especially on interactive applications, which perform more data exchange with remote servers.

4) PERFORMANCE OF QUICK SERVICE PROVISIONING

We study the performance of quick service provisioning (QSP, details in Section V-D) by measuring the service provisioning time, which is defined as a period between receiving an offloading request and being ready to run the task. Specifically, it includes the time of VM initialization and user execution environment setup. As shown in Fig. 12, QSP can significantly reduce service provisioning time due to quick VM synthesis and application package pre-installation.

C. TRACE-DRIVEN SIMULATIONS

We develop a trace-driven simulator to evaluate Echo at a larger scale. First, we collect traces from mobile users in the real deployment, who run five applications with different operations. For each annotated offloading method, we measure its running time at different platforms as well as the data uploading and downloading time. Then, we choose three sets of traces by mixing five applications with different proportions. Each set contains 100 offloading requests. The first set of traces (trace ID 1) contains 80% methods from interactive applications (listed in Table 2) and 20% methods...
from compute-intensive ones. In the second set of traces (trace ID 2), we reverse the proportion by including 20% interactive applications and 80% compute-intensive ones. The proportion is 50% for both in the third set (trace ID 3). We use a widely-adopted service model that the interval time between two requests is an exponential distribution with parameter $\lambda$, so that the number of requests in unit time is a normal distribution. In our simulations, there are 100 offloading requests and the number of VMs at the edge node can be changed from 1 to 8.

We first set $\lambda = 1$ and show the results in Fig. 13. It is obviously that Echo outperforms other systems in all cases, and their performance gap increases as more VMs are available at the edge. However, we observe that the performance gap between Echo and mCloud becomes smaller when the number of VMs grows. That is because more VMs mitigate resource contention at the edge, and the efficiency of our scheduling algorithm PC-SRTF is not so obvious. We then increase resource contention by setting $\lambda = 2$, i.e., the mean of time interval between two requests is 0.5 second. As shown in Fig. 14, Echo brings more benefits. For example, when there are 8 VMs, Echo reduces the average completion time by 24.3% under the first set of traces compared with mCloud, while the reduction is only 17.4% when $\lambda = 1$.

![Average completion time when $\lambda = 2$. (a) 1 VM. (b) 4 VMs. (c) 8 VMs.](image-url)

VIII. CONCLUSION

In this paper, we propose Echo, an edge-centric code offloading system over mobile devices, the edge, and the cloud with QoS guarantee. It has a centralized decision engine that collects offloading requests from mobile devices and decides which methods should be offloaded to the edge or the cloud. To reduce the average task completion time, we propose a heuristic algorithm called PC-SRTF for task scheduling at edge nodes. Echo also optimizes network transmission by lazy object transmission and differential object update. Through a small-scale real deployment and trace-driven simulations, we show that Echo can significantly outperform existing code offloading systems.

REFERENCES

[1] E. Cuervo et al., “MAUI: Making smartphones last longer with code offload,” in Proc. 8th Annu. Int. Conf. Mobile Syst., Appl., Services (MobiSys), Mar. 2010, pp. 49–62.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic execution between mobile device and cloud,” in Proc. 6th Eur. Conf. Comput. Syst. (EuroSys), Apr. 2011, pp. 301–314.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in Proc. 31st IEEE Int. Conf. Comput. Commun. (INFOCOM), Mar. 2012, pp. 945–953.

[4] M. Satyanarayanan, “The emergence of edge computing,” Computer, vol. 50, no. 1, pp. 30–39, Jan. 2017.

[5] P. G. Lopez et al., “Edge-centric computing: Vision and challenges,” ACM SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, Sep. 2015.

[6] S. Teerapittayanon, B. McDaniel, and H. T. Kung, “Distributed deep neural networks over the cloud, the edge and end devices,” in Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS), Jun. 2017, pp. 328–339.

[7] S. Nastic, H. L. Truong, and S. Dustdar, “A middleware infrastructure for utility-based provisioning of IoT cloud systems,” in Proc. 1st IEEE/ACM Symp. Edge Comput. (SEC), Oct. 2016, pp. 28–40.

[8] K. Lee, J. Flinn, and B. D. Noble, “Grenlin: Scheduling interactions in vehicular computing,” in Proc. 2nd ACM/IEEE Symp. Edge Comput. (SEC), Oct. 2017, pp. 4:1–4:13.

[9] I.-H. Hou, T. Zhao, S. Wang, and K. Chan, “Asymptotically optimal algorithm for online reconfiguration of edge-clouds,” in Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput. (MobiHoc), Jul. 2016, pp. 291–300.

[10] L. Tong, Y. Li, and W. Gao, “A hierarchical edge cloud architecture for mobile computing,” in Proc. 35th IEEE Int. Conf. Comput. Commun. (INFOCOM), Apr. 2016, pp. 1–9.

[11] M. S. Gordon, D. A. Janshidi, S. Mahlke, Z. M. Mao, and X. Chen, “COMET: Code offload by migrating execution transparently,” in Proc. 10th USENIX Symp. Operat. Syst. Design Implement (OSDI), Oct. 2012, pp. 93–106.

[12] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Siramam, and R. Buyya, “mCloud: A context-aware offloading framework for heterogeneous mobile cloud,” IEEE Trans. Services Comput., vol. 10, no. 5, pp. 797–810, Sep. 2017.

[13] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” IEEE Pervasive Comput., vol. 8, no. 4, pp. 14–23, Oct. 2009.

[14] A. Bhattacharya and P. De, “A survey of adaptation techniques in computation offloading,” J. Netw. Comput. Appl., vol. 78, pp. 97–115, Jan. 2017.

[15] X. Jin, Y. Liu, W. Fan, F. Wu, and B. Tang, “Multisite computation offloading in dynamic mobile cloud environments,” Sci. China Inf., vol. 60, no. 8, p. 089301, Feb. 2017.

[16] G. Liang, J. Ca, X. Liu, and J. Liang, “Smart world: a better world,” Sci. China Inf., vol. 59, no. 4, p. 043401, Feb. 2016.
[17] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng, “AnySee: Peer-to-peer live streaming,” in Proc. 25th IEEE Int. Conf. Comput. Commun. (INFOCOM), Apr. 2006, pp. 1–10.

[18] X. Chen, L. Jiao, W. Li, and X. Fu, “Efficient multi-user computation offloading for mobile-edge cloud computing,” IEEE/ACM Trans. Netw., vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[19] M. Jia, J. Cao, and W. Liang, “Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks,” IEEE Trans. Cloud Comput., vol. 5, no. 4, pp. 725–737, Oct. 2017.

[20] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in Proc. ACM Int. Conf. Mobile Syst., Appl., Services (MobiSys), Jun. 2014, pp. 68–81.

[21] K. Ha and M. Satyanarayanan, “OpenStack++ for cloudlet deployment,” Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., 2015.

[22] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, “Container-based cloud platform for mobile computation offloading,” in Proc. 31st IEEE Int. Parallel Distrib. Process. Symp. (IPDPS), May 2017, pp. 123–132.

[23] S. Wu, C. Mei, H. Jin, and D. Wang, “Android unikernel: Gearing mobile code offloading towards edge computing,” Future Gener. Comput. Syst., vol. 86, pp. 694–703, Sep. 2018.

[24] X. Wang et al., “Dynamic resource scheduling in mobile edge cloud with cloud radio access network,” IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 11, pp. 2429–2445, Nov. 2018.

[25] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan, “A scalable and privacy-aware IoT service for live video analytics,” in Proc. 8th ACM Multimedia Syst. Conf. (MMSys), Jun. 2017, pp. 38–49.

[26] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, “Glimpse: Continuous, real-time object recognition on mobile devices,” in Proc. 15th ACM Conf. Embedded Netw. Sensor Syst. Pages (SenSys), Nov. 2015, pp. 155–168.

[27] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, “Just-in-time provisioning for cyber foraging,” in Proc. 11th Annu. Int. Conf. Mobile Syst., Appl., Services (MobiSys), Jun. 2013, pp. 153–166.

[28] D. Narayanan, J. Flinn, and M. Satyanarayanan, “Using history to improve mobile application adaptation,” in Proc. 3rd IEEE Workshop Mobile Comput. Syst. Appl. (MCSA), Dec. 2000, pp. 31–40.

[29] S. Gurun, C. Krintz, and R. Wolinski, “NWSLine: A light-weight prediction utility for mobile devices,” in Proc. 2nd Annu. Int. Conf. Mobile Syst., Appl., Services (MobiSys), Jun. 2004, pp. 2–11.

[30] S. Arzt et al., “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI), Jun. 2014, pp. 259–269.

[31] M. R. Hines, U. Deshpande, and K. Gopalani, “Post-copy live migration of virtual machines,” SIGOPS Oper. Syst. Rev., vol. 43, no. 3, pp. 14–26, Jul. 2009.

[32] P. Sved, B. Hudzja, J. Tordsson, and E. Elmroth, “Evaluation of delta compression techniques for efficient live migration of large virtual machines,” in Proc. 7th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE), Mar. 2011, pp. 111–120.

[33] P. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in Proc. 22nd ACM SIGPLAN-SIGACT Symp. Principles Program. Lang. (POPL), Jan. 1995, pp. 49–61.

[34] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The Soot framework for java program analysis: A retrospective,” in Proc. 1st Cetus Users Compiler Infrastructure Workshop (CETUS), Oct. 2011, p. 35.

[35] Z. Chen et al., “An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance,” in Proc. 2nd ACM/IEEE Symp. Edge Comput. (SEC), Oct. 2017, pp. 14:1–14:14.

[36] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, “A close examination of performance and power characteristics of 4G LTE networks,” in Proc. 10th Annu. Int. Conf. Mobile Syst., Appl., Services (MobiSys), Jun. 2012, pp. 225–238.

[37] L. Zhang et al., “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in Proc. 8th IEEE/ACM/FIP Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS), Oct. 2010, pp. 105–114.