# FUNOff: Offloading Applications at Function Granularity for Mobile Edge Computing

Xing Chen, *Member, IEEE*, Ming Li, Hao Zhong, *Member, IEEE*, Xiaona Chen, Yun Ma, *Member, IEEE*, and Ching-Hsien Hsu, *Senior Member, IEEE*

***Abstract*—Mobile edge computing (MEC) offers a promising technology that deploys computing resources closer to mobile devices for improving performance. Most of the existing studies support on-demand remote execution of the computing tasks in applications through program transformation, but they commonly assume that mobile devices merely resort to a single server for computation offloading, which cannot make full use of the scattered and changeable computing resources. Thus, for object-oriented applications, we propose a novel approach, called FUNOff, to support the dynamic offloading of applications in MEC at the function granularity. First, we extract a call tree via code analysis and locate the function invocations that are suitable for offloading. Next, we refactor the code of related object functions according to a specific program structure. Finally, we make offloading decisions referring to the context at runtime and send function invocations to multiple remote servers for execution. We evaluate the proposed FUNOff on two real-world applications. The results show that, compared with other approaches, FUNOff better supports the computation offloading of object-oriented applications in MEC, which reduces the response time by 10.7%-58.2%.**

***Index Terms*—Mobile edge computing, computation offloading, code analysis, object-oriented application, software adaptation.**

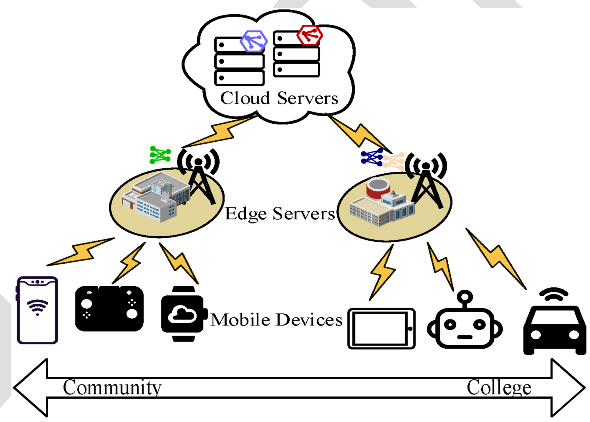



Fig. 1. Mobile edge computing (MEC) architecture.

## I. Introduction

With the rise of intelligent technologies, massive computation-intensive applications (e.g, autonomous driving [1], image recognition [2], and augmented reality [3]) have been developed to improve the quality of people's ~~life.~~ However, most existing smart devices (e.g., wearable devices [4], vehicles [5], and UAVs [6]) are unable to handle computation-intensive tasks in a short time due to the constraints on their processing power, memory capacity, and battery capacity [7].

Computation offloading is an effective way to resolve resource constraints on mobile devices [8]. In the last decade, one feasible way is to offload computation-intensive tasks from mobile devices to a cloud server, aiming to improve the performance of mobile applications [9], [10], [11]. This paradigm is known as mobile cloud computing (MCC). Although MCC elevates user experience, higher network delay can happen, if the cloud server is remote [12]. Meanwhile, the massive data transmission between the cloud server and mobile devices increases the traffic load of core networks [13]. When there are many mobile devices, the performance of MCC may be seriously affected, especially for latency-sensitive applications. To further improve MCC, a new paradigm, called mobile edge computing (MEC) has emerged. Fig. 1 depicts a typical MEC architecture: there is a three-tier computing architecture consisting of mobile devices, edge nodes, and the cloud [14], [15]. By pushing the computing resources from the centralized cloud to the decentralized edges near the data source (e.g., mobile devices), MEC reduces the influx of data on the backbone [16], [17]. Therefore, MEC has been regarded as a more effective way to reduce ~~the service delay.~~ than MCC does.

Due to the geographical distribution of MEC servers and the mobility of mobile devices, the runtime context in MEC is highly complex and dynamic [18], [19]. Although the prior studies [20], [21], [22] can be extended to the scenario of MEC, they lack enough effectiveness, since they only divide an application into

Manuscript received 11 May 2022; revised 11 January 2023; accepted 19 January 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62072108 and in part by the Natural Science Foundation of Fujian Province for Distinguished Young Scholars under Grant 2020J06014. Recommended for acceptance by Dr. J. Wu. *(Corresponding author: Hao Zhong.)*

Xing Chen, Ming Li, and Xiaona Chen are with the College of Computer and Data Science, Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou 350118, China (e-mail: chenxing@fzu.edu.cn; 210310011@fzu.edu.cn; cxn_95@163.com).

Hao Zhong is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: zhonghao@sjtu.edu.cn).

Yun Ma is with the School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: mayun@pku.edu.cn).

Ching-Hsien Hsu is with the Department of Computer Science and Information Engineering, Asia University, Taichung 413, Taiwan (e-mail: robertchh@gmail.com).

Digital Object Identifier 10.1109/TMC.2023.3240741

two parts and deploy them on a mobile device and a remote server. In our prior work, we propose an adaptive offloading architecture, called Androidoff [23], [24]. Androidoff is able to offload applications among the local device, mobile edges, and the cloud dynamically, but it still ~~reveals~~ the following limitations:

(1) There is still improvement space for the performance of Androidoff. The Androidoff offloads applications at the granularity of objects, but it would be more flexible by adopting finer granularity. For example, an object owns two methods, which intend to be offloaded to the edge and the cloud, respectively. However, since these two methods are from the same object, they can only be offloaded to the same location (i.e., the edge or the cloud).

(2) When users move to a new location, Androidoff ensures the normal operation of applications by accessing the copied objects of the cloud server. If the new environment is not connected to the cloud server, some information may be lost, which causes crashes. Meanwhile, the time of restarting applications is often unacceptable.

Although it is beneficial to offload applications at a finer granularity, it is challenging to decompose applications. Most applications are monolithic and have a high degree of internal coupling [25]. Moreover, another challenge is to avoid loss of information when users move to new scenarios. Mobile devices need to maintain all the state information of the objects to ensure that the application can keep executing normally.

Recently, the Function as a Service (FaaS) programming model has been widely adopted with the emergence of serverless cloud computing [26], [27]. In FaaS, an application is split into short-lived stateless functions that can be executed by different computing nodes [28], which is a fine-grained computation offloading. The basic idea of FaaS can resolve the problem of information loss caused by a finer granularity. However, to realize this idea, there are two key challenges: (1) The execution of a function in an object-oriented (OO) application depends on the states of multiple objects. (2) To adapt to the highly complex and dynamic runtime context of MEC, an algorithm shall make quick offloading decisions.

To address the problems of the state-of-the-art, we propose a novel offloading mechanism, called FUNOff. The major contributions of this paper are as follows:

- A novel offloading mechanism, called FUNOff, that supports the offloading of applications at the granularity of functions. The FUNOff builds a call tree, and discovers function invocations that are suitable for offloading. To resolve the state dependencies of methods, the FUNOff transforms functions into stateless ones based on the code analysis results.
- An online decision traversal strategy that uses the properties of the call tree and the tendency of computation offloading to synthesize offloading schemes.
- Extensive evaluation results on two real-world applications. We evaluate FUNOff on License Plate Recognition Application (LPRA) and Target Detection Application (TDA). Compared with the existing approaches [9], [23], [24], the results show that FUNOff reduces the response time of LPRA and TDA by 10.7%-45.7% and 14.5%-58.2%, respectively.

## II. Related Work

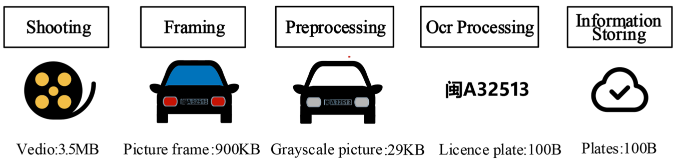
### A. Offloading Mechanism

Computation offloading is a way to resolve resource constraints on mobile devices. The state-of-the-art offloading mechanism can offload applications by the granularity of program fragments [11], methods [9], [20], [21], classes [29], layers [22], and objects [23], [24].

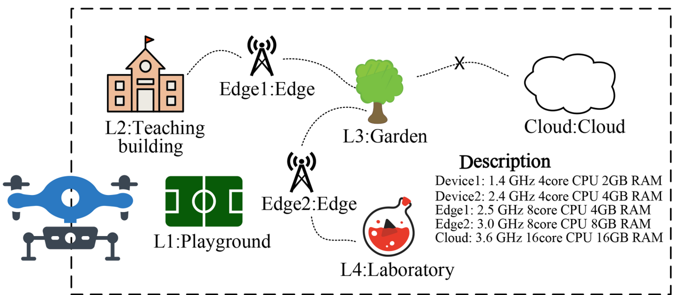
Cuckoo [11] is a computation offloading framework with the granularity of program fragments. It asks developers to comply with a given programming paradigm to refactor the application so that certain parts of it can be offloaded to the cloud server. DPartner [29] can offload classes, and it uses a proxy mechanism to access class instances. Further, it calculates the coupling of classes and deploys them in two parts on a mobile device or a remote cloud server. Although the above approaches can effectively support computation offloading of applications in MCC, they are not designed for MEC. MAUI [9] is a computation offloading framework for C# applications, which offloads applications at the granularity of methods. The programmers only need to mark remoteable methods, and the application can be restructured automatically. Then the framework will decide which methods should be offloaded to the remote server at runtime. ULOOF [20] also works on the granularity of methods, but it targets the offloading problem for Java applications. Dandelion [21] is a unified code offloading system for wearable computing that supports multi-process offloading. It can generically offload tasks to a cloud, a cloudlet, or nearby smart devices. DeepWear [22] strategically offloads DL tasks from a wearable device to its paired handheld device. It splits a DL model into two sub-models that are first executed on the wearable and then on the handheld. However, the above studies only divide the application into two parts and deploy them on a mobile device and a remote server, respectively. This paradigm cannot support the dynamic offloading among the device, mobile edges, and the cloud [30], [31], [32], which limits performance improvement. To address this issue, AndroidOff [23], [24] proposed an adaptive offloading framework that supports computation offloading at the object granularity in MEC. It enables offloading applications among the local device, mobile edges, and the cloud dynamically. However, the stateful nature of the methods makes AndroidOff inapplicable in some scenarios.

### B. Offloading Strategy

Computation offloading needs to determine which parts of an application shall be offloaded and to which compute nodes, i.e., the decision of an offloading scheme. A qualified offloading scheme needs to balance the impact of various factors, such as computing performance and network environment, around the offloading goal. In recent years, researchers have started to explore the intelligent scheduling of computation offloading in MCC or MEC.




(a) Process of a license plate recognition application




(b) Context of the drone in the college

Fig. 2. The sample scenario (a) Process of a license plate recognition application. (b) Context of the drone in the college.

Altamimi et al. [33] evaluated the communication energy consumption of offloading computing tasks to cloud servers and established a high-precision energy consumption estimation model without the requirement of complete input parameters. It can decide whether computing tasks shall be offloaded based on this model rapidly. Elgazzar et al. [34] proposed a framework for collaborative offloading services to provide computation offloading services for mobile devices based on the system network, resource status, and energy consumption constraints. Zhou et al. [35] proposed a context-aware offloading decision algorithm to provide offloading decisions at runtime, called mCloud, which selects a wireless medium and appropriate cloud resources for offloading. The works [33], [34], [35] aim at intelligent scheduling in MCC, and some works [30], [36] are proposed for MEC. Cheng et al. [30] proposed a three-layer computation offloading framework composed of wearable devices, mobile devices, and edge nodes. They introduced genetic algorithms to increase the task throughput of wearable devices in MEC. Wu et al. [36] proposed a task partition algorithm suitable for the computation offloading of graph applications. They adopted an improved bipartite graph algorithm to divide the computing tasks into local and remote ones. However, the above approaches make offloading decisions based on the high-level abstract model of a program, rather than a real application.

## III. Motivation

MAUI [9] is a well-known computation offloading framework, which supports the dynamic offloading of object-oriented programs at method granularity in MCC. It allows annotating which methods can be offloaded beforehand and deciding the offloading scheme at runtime. AndroidOff [23], [24] is an adaptive offloading framework for MEC. It is designed to handle object-oriented programs and offload them at the object granularity. In this section, we use a scene as shown in Fig. 2 to illustrate how MAUI [9], AndroidOff, and FUNOff work. In this scene, the drone cruises around the college, and when it detects illegal parking, the LPRA in the drone will be operated to identify the car's plate number from the video stream. Fig. 2(a) shows the process of LPRA, including shooting, framing, preprocessing, ocr processing, and information storing. Each process contains several functions, as shown in Fig. 7(a). These tasks require different computation power. For example, ocr processing is a computation-intensive task, and it is more effective to offload it to a remote server; meanwhile, framing exhibits low computation complexity. The data traffic between tasks is another influencing factor. For example, the data traffic between shooting and framing is large, while between preprocessing and ocr processing is marginal. It is preferred to execute two adjacent tasks with high data traffic on the same device. Fig. 2(b) shows the context of the drone when it cruises around the college. There are three available remote servers (Cloud, Edge1, and Edge2) in different locations. Edge1 is located in the teaching building and the garden; Edge2 can be accessed from the garden and the laboratory; Cloud can be accessed from other locations besides the garden. Notes that the network environment and the LPRA are the same as the setting in Section V. To improve the performance, when the drone stays in different locations, it needs to determine where each computation task is executed and then offload each task to its corresponding server in a real-time manner. When the drone moves to a different location, its application must be smoothly switched between servers.

We discuss two offloading cases:

Case 1: When the drone stays in a location, it must be able to utilize the scattered computing resources around the location. For example, the drone can use a cloud server and an edge server to improve the performance of LPRA in the Laboratory. In this location, FUNOff offloads computation-intensive functions such as RecInEachChar.getHZ() and Oritenation.math() to the cloud or edge by comparing the reduced execution time with the increased network latency. If functions implement simple tasks, they are executed locally. As for MAUI only uses a single remote server for computation offloading due to its poor scalability. It cannot offload different methods to multiple different remote servers to further enhance performance, so RecInEachChar.getHZ() and Oritenation.math() are both offloaded to the edge server. As a result, MAUI can only reduce the response time by 34%, while FUNOff can reduce it by 46%. According to our offloading scheme, getHZ() and GetRegion() are executed on the cloud and drone, respectively. Androidoff is offloaded at object granularity, getHZ() and GetRegion() can only be offloaded to the cloud since they are both methods of object RecInEachChar.

Case 2: When the drone moves between different locations, it shall switch smoothly. For example, suppose that the drone moves from the teaching building to the garden. In the beginning, the drone executes the LPRA in the teaching building. It offloads the function RecInEachChar.getHZ() to Cloud according to our offloading scheme. During the application execution, the drone moves to the garden, causing the application to disconnect from Cloud. Since both FUNOff and MAUI save the information of the object RecInEachChar in the drone, they can ensure the normal execution of RecInEachChar.getHZ() in the new context.

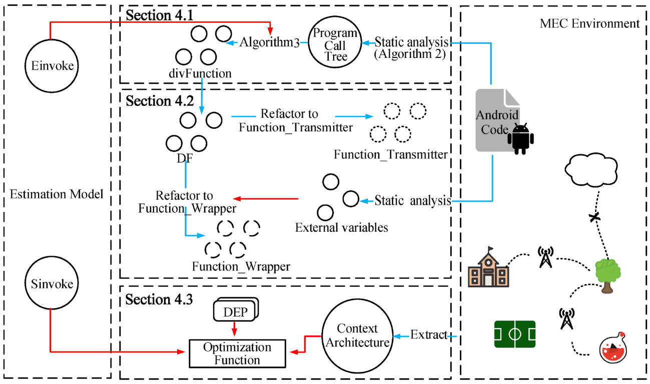



Fig. 3. The overview of FUNOff.

As Androidoff takes the object as the minimum offloading unit, the state information of the object is saved on the corresponding execution location. The application will crash when the drone moves from the teaching building to the garden. The information of the object RecInEachChar is not saved on the drone, and there is no connection to the Cloud to get the information. As a result, the crash caused $20s$ delay in restarting the application. In order to make offloading smoother, the generation time of offloading schemes needs to be reduced. FUNOff can reduce it by determining the appropriate cut-point functions in advance.

Compared with the above approaches, FUNOff has the following main improvements: (1) it can support adaptive offloading at function granularity in MEC; (2) The object methods are translated into stateless functions to avoid the loss of state information caused by movement. (3) To support offloading at runtime, the set of cut-point functions suitable for offloading is automatically determined in advance to reduce the generation time of the offloading scheme.

## IV. Approach

Fig. 3 shows the overview of FUNOff. The FUNOff reuses the estimation model of AndroidOff proposed in our previous work [23]. This model predicts the execution costs (i.e., execution time) of functions. Based on this model, FUNOff further introduces a code analyzer (Section IV-A), an offloading mechanism (Section IV-B), and an offloading strategy (Section IV-C). These components interact with an MEC environment.

More specifically, Algorithm 1 gives the details. It takes the source code of an application and an MEC environment as its input. Here, the MEC environment is modeled as a graph, in which nodes represent computing nodes (including the mobile device and remote servers with different computation capabilities), and edges represent the communication link between two computing nodes (e.g., the data transmission rate and round-trip time). The output of Algorithm 1 is the offloading scheme, which includes the execution location of each function in the call tree. Algorithm 1 includes the following three procedures:

Procedure 1 (Section IV-A): We implement a code analyzer to extract suitable function invocations. First, it builds a call tree. In this tree, the entry is the main() function; each node represents a function; and a directed edge between nodes represents a function call between two functions (Line 2). Based on the computation complexity and data transmission of each function, it extracts function invocations that are suitable for offloading (Line 3).

TABLE I
Symbol and Description

| Symbol | Description |
|---|---|
| $Tree_{f_r} = (F, R)$ | Call tree beginning at $f_r$, where $F$ denotes the set of functions, and $R$ denotes the set of call relations between functions |
| $fSignature_i$ | Function signature of $f_i$ |
| $callSeq_i$ | Function call path from $f_{main}$ to $f_i$ |
| $r_{i-j} \in R$ | Function call from $f_i$ to $f_j$ |
| $U_j$ | Soot statement set of $f_i$ |
| $u_j^i$ | $i$-th soot statement of $U_j$ |
| $N$ | Set of computing nodes, including $DS$, $ES, CS$ |
| $n_k \in N$ | Computing node $n_k$ |
| $v_{n_p - n_q}$ | Data transmission rate between $n_p$ and $n_q$ |
| $rtt_{n_p - n_q}$ | Round-trip time between $n_p$ and $n_q$ |
| $dep\left(f_i\right)$ | Execution location of $f_i$ |
| $T_{response}$ | Response time of application |
| $T_{dep(f_j)}^{dep(f_i)}\left(f_i\right)$ | Total offloading time of $f_i$ |
| $T_e{}^{dep(f_i)}\left(f_i\right)$ | Execution time of $f_i$ in $dep\left(f_i\right)$ |
| $T_d{}_{dep(f_j)}^{dep(f_i)}\left(f_i\right)$ | Data transmission time of $f_i$ |
| $Einvoke_{n_q}^{f_i}$ | Execution cost of $f_i$ on $n_q$ |
| $Sinvoke_{n_q}^{f_i}$ | Execution cost except external invocations of $f_i$ on $n_q$ |

Procedure 2 (Section IV-B): We implement an offloading mechanism to enable the remote calls of functions. For the functions extracted in Procedure 1, Line 6 extracts their signatures, and Lines 7 to 11 construct wrappers and transmitters for them according to our program structure.

Procedure 3 (Section IV-C): Based on the results of the above procedures, we design an offloading strategy to determine the offloading scheme according to the context automatically. Different parts of the application can be executed on mobile devices, edge servers, or cloud servers. With this offloading strategy, we implemented an offloading decision algorithm (Algorithm 4). For an application, this algorithm uses the optimization function to calculate the response time of each candidate offloading scheme and selects the scheme with the minimum value.

Table I lists the major symbols used in this paper.

### A. Code Analyzer

As only a few function calls are suitable for offloading, we employ a preprocessing step, i.e., a program analysis technique for computing offloading. We extract a call tree through static analysis (Section IV-A1). After that, we identify the function invocations suitable for offloading (Section IV-A2) to reduce the additional execution cost of the offloading mechanism and the time cost caused by the decision of offloading schemes.

*1) Extracting the Call Tree:* FUNOff builds a call tree for an object-oriented application. The definition of the call tree is as follows:

*Definition 1.* $Tree_{f_r} = (F, R)$ denotes a call tree beginning at $f_r$, where $F = \{f_1, f_2, \ldots, f_n\}$ is the set of function call sites,

**Algorithm 1:** The Overview of FUNOff.

**Input:** The source code of an application $code$; A context environment $G_c = (N, E)$
**Output:** the offloading scheme $(DEP)_{optimal} = \{dep(f_1), \ldots, dep(f_n)\}$ and the response time $(T_{response})_{optimal}$

**procedure** 1
  $Tree_{f_{main}} = (F, R) \leftarrow$ Algorithm 2($code$)
  $divFunction = \{f_1, f_2, \ldots, f_p\} \leftarrow$ Algorithm 3($Tree_{f_{main}}$)
**end procedure**
**procedure** 2
  organize $divFunction$ as $DF$
  **for** each $df_i \in DF$ **do**
    $Param_i \leftarrow$ collect external parameters of $df_i$
    $df_i\_Wrapper \leftarrow$ refactor $df_i$ with $Param_i$
    $df_i\_Transmitter \leftarrow$ refactor $df_i$
  **end for**
**end procedure**
**procedure** 3
  $\langle (DEP)_{optimal}, (T_{response})_{optimal} \rangle \leftarrow$ Algorithm 4($Tree_{f_{main}}, G_c, Sinvoke$)
**end procedure**

**Algorithm 2:** Extracting the Call Tree.

**Input:** A $f_{main}$ function whose statements are $\{u^1_{main}, \ldots, u^n_{main}\}$
**Output:** A call tree $Tree_{f_{main}} = (F, R)$

1: $F \leftarrow F + f_{main}, R \leftarrow \emptyset$
2: **function** getTree$f_a, U_a$
3:   **for** each $u^i_a \in U_a$ **do**
4:     $keywords \leftarrow Soot(u^i_a)$
5:     **if** $\exists''invoke'' \in keywords$ **then**
6:       $fSignature \leftarrow$ getfunction($u^i_a$)
7:       $callSeq \leftarrow f_a.callSeq + fSignature$
8:       $f_s \leftarrow \langle fSignature, callSeq \rangle$
9:       $U_s \leftarrow$ getUnits($fSignature$)
10:       $F \leftarrow F + f_s$
11:       **if** $\langle f_a, f_s \rangle \in R.key$ **then**
12:         $++r_{f_a - f_s}$
13:       **else**
14:         $r_{f_a - f_s} \leftarrow 1$
15:         $R \leftarrow R + r_{f_a - fs}$
16:       **end if**
17:       getTree($f_s, U_s$)
18:     **end if**
19:   **end for**
20: **end Function**

and $R$ is the set of function call relations. Each edge $r_{i-j} \in R$ represents a function call from $f_i$ to $f_j$, and its weight represents the call times of the function call.

*Definition 2.* $f_i = \langle fSignature_i, callSeq_i \rangle, f_i \in F$: $fSignature_i$ denotes function signature of $f_i$, and $callSeq_i$ denotes a function call path from the main() function (denoted as $f_{main}$, the same below) to $f_i$.

FUNOff uses Soot[1] to build call trees, and Algorithm 2 shows its details. It takes $f_{main}$ as the entry of the application, and extracts the call tree beginning at $f_{main}$. We get a hash map to record $R$, whose keys are stored in the form of $\langle f_i, f_j \rangle$, and its corresponding value means the times of the call from $f_i$ to $f_j$. The inputs of Algorithm 2 are the entry function $f_{main}$ and its soot statement set $U_{main}$, each $u^i_{main}$ denotes the *i*th soot statement of $U_{main}$. Lines 3 to 20 extract the call tree recursively via the function getTree(), its parameters $f_a$ denotes the function to be analyzed, and $U_a$ denotes $f_a$'s soot statements. In particular, Line 4 obtains the soot keywords in $u^i_a$, which is the instructions defined in Soot. For example, the *invoke* keyword indicates a function call statement. The complete keywords are defined in the Soot manual[1]. Therefore, if $u^i_a$ contains a keyword that indicates a call to function $f_s$, Lines 6 to 10 update $F$, that is, add $f_s$ to set $F$. Lines 11 to 16 update $R$, that is, record the function call from $f_a$ to $f_s$ and update its corresponding value. Line 17 recursively calls the function getTree() with $f_s$ and its statement set. When the procedure is done, the call tree is obtained.

*2) Extracting Cut-Point Functions:* According to the call tree extracted in Section IV-A1, FUNOff further identifies function invocations that are suitable for offloading. For the convenience of description, we call such function invocations cut-point functions. Table II shows the factors that are collected to identify cut-point functions. We estimate the performance ratio between the computing nodes according to the ratio of the time required to process a set of identical functions on these nodes. The estimation model of AndroidOff [23] is able to predict the execution costs of all functions. Following its definition, we use $Einvoke^{f_i}_{n_q} = \langle Etime, Edatasize \rangle$ to denote the execution cost of function $f_i$ at the computing node $n_q$, where $Etime$ denotes the execution time, and $Edatasize$ denotes the amount of data transmission.

TABLE II
FACTORS FOR IDENTIFYING CUT-POINT FUNCTIONS

| Symbol | Description |
|---|---|
| $\lambda$ | Performance ratio between the remote computing node and the local computing node |
| $v$ | Transmission rate between the remote computing node and the local computing node |
| $rtt$ | Round-trip time between the remote computing node and the local computing node |

For each branch path with the current node as the starting node, get all nodes on the path from the current node to the first branch node, and FUNOff chooses the cutpoint functions from them.

In particular, $Tree_{f_{cur}}$ denotes the subtree rooted at the function $f_{cur}$ of the call tree, and $T^{Tree_{f_{cur}}}$ denotes the response time of its local execution. $T^{Tree_{f_{cur}}}(f_i)$ denotes the response

[1] https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/

time of $Tree_{f_{cur}}$ on the $f_i$ function call and is calculated as the (1). If $f_i$ is identified as a cut-point function, all the functions in the subtree rooted at $f_i$ can be executed on a remote computing node. The response time consists of the local execution time, the remote execution time, and the data transmission time.

Eq. (2) denotes the local execution time, which is calculated as the difference between the total execution time of $f_{cur}$ and that of $f_i$ on the local computing node. In particular, $r_{f_i.caller-f_i}$ donates the call times of $f_i$ in a function call to $f_{cur}$, it calculates as the product of weight on the path from $f_{cur}$ to $f_i$.

Eq. (3) denotes the remote execution time, which is quantified by the execution cost on a remote computing node over that of a local one.

Eq. (4) denotes the total data transmission time. It consists of the transmission time and the round-trip time. In particular, the transmission time is the amount of data transmission of the cut-point function divided by the transmission rate between the remote computing node and the local computing node, and the round-trip time between two computing nodes is represented as $rtt$.

FUNOff extracts cut-point functions based on the above rules and equations. Algorithm 3 describes the details, where the input is the call tree $Tree_{f_{cur}}$ of the program, and the output is the set of cut-point functions $divFunction$. Line 1 sets the $divFunction$ to the empty set. Then, Line 2 takes $f_{main}$ as the current function $f_{cur}$ of the call tree and uses the $GetDivFunction()$ function to get the $divFunction$ recursively. Lines 3 to 23 of Algorithm 3 describe the details of the $GetDivfunction()$. Line 4 checks whether $f_{cur}$ has a successor. If it has, Lines 5 to 21 do the following operations on each branch: Lines 7 to 8 add the functions on this branch path to the set $P$ in order until the first branch node is found. If it exists, line 10 takes it as the current function and recursively calls $GetDivFunction()$. Lines 14 to 15 iterate through the functions in $P$ in turn until a function $f_i$ is found, so that the response time of $Tree_{f_{cur}}$ on the $f_i$ function call is less than the time of local execution. After that, lines 16 to 17 add $f_i$ to $divFunction$, and call the function $GetDivFunction()$ recursively. When Algorithm 3 is done, a set of all cut-point functions is obtained.

**Algorithm 3:** Extracting Cut-Point Functions.

**Input:** A call tree $Tree_{f_{main}} = (F, R)$
**Output:** A set of cut-point functions $divFunction = \{f_1, f_2, \ldots, f_n\}$

```
1: divFunction ← ∅
2: getDivfunction(f_main)
3: function getDivfunction f_cur
4:   if post(f_cur) ≠ ∅ then
5:     for each branch path do
6:       P ← ∅
7:       for each f_i in this branch path except f_cur do
8:         P ← P ∪ f_i
9:         if f_i is a branch node then
10:          getDivfunction(f_i)
11:          break
12:        end if
13:      end for
14:      for each f_i in P do
15:        if T^{Tree_fcur}(f_i) < T^{Tree_fcur} then
16:          divFunction ← divFunction ∪ f_i
17:          getDivfunction(f_i)
18:          break
19:        end if
20:      end for
21:    end for
22:  end if
23: end Function
```

Under the MEC environments with various computational resources and network connections, there might be different numbers of functions that are suitable to be offloaded. Basically, the offloading tends to happen when the higher performance ratio ($\lambda$) between servers and IoT devices and faster data transmission rate ($v$ and $rtt$). In practical applications, we select the performance ratio, network transmission rate, and round-trip time between the remote and the local computing nodes of the optimal offloading environment in the current scenario as $\lambda$, $v$, and $rtt$ to avoid missing the necessary cut-point functions. With these factors, the optimization function extracts cut-point functions that are suitable for offloading and deploying them to different computing nodes.

In an offloading problem, the decision time of offloading strategy is linearly positive to the number of functions in an application, and finding the cut-point functions in advance can effectively reduce the decision time.

### B. Offloading Mechanism

A standalone application typically is designed to execute on only a mobile device. To enable its offloading, FUNOff modifies

$$T^{Tree_{f_{cur}}}(f_i) = T_e{}^{Tree_{f_{cur}}}(f_i)[local] + T_e{}^{Tree_{f_{cur}}}(f_i)[remote] + T_d{}^{Tree_{f_{cur}}}(f_i) \tag{1}$$

$$T_e{}^{Tree_{f_{cur}}}(f_i)[local] = Einvoke^{f_{cur}}_{n_{cur}}.Etime - Einvoke^{f_i}_{n_{cur}}.Etime * r_{f_i.caller-f_i} \tag{2}$$

$$T_e{}^{Tree_{f_{cur}}}(f_i)[remote] = \frac{Einvoke^{f_i}_{n_{cur}}.Etime * r_{f_i.caller-f_i}}{\lambda} \tag{3}$$

$$T_d{}^{Tree_{f_{cur}}}(f_i) = \left(\frac{Einvoke^{f_i}_{n_{cur}}.Edatasize}{v} + rtt\right) * r_{f_i.caller-f_i}. \tag{4}$$

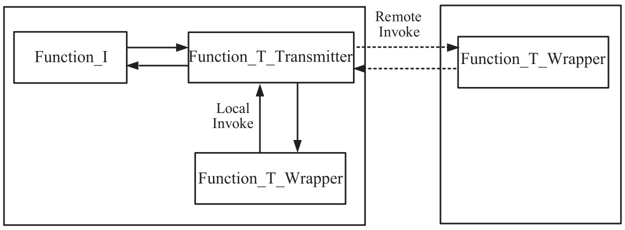



Fig. 4. Target program structure.

```
1:    void function (int a, int b) {
2:            d = a +b +c;
3:            return null;
4:    }
```

(a) The source code

```
1:    Result function (Params params, int a, int b) {
2:            params.d = a +b + params.c;
3:            Result result = new Result();
4:            result.params = params;
5:            result.val = null;
6:            return result;
7:    }

Params {
    int c,d;    //External variables
}

Result {
    Params params;    //Additional return value
    Object val;    //Original return value
}
```

(b) The target code of function wrapper

Fig. 5. Example of original function and function wrapper, where $c$ and $d$ were external variables (a) The source code. (b) The target code of function wrapper.

the source files of applications. To keep program behaviors unchanged, FUNOff builds wrappers for stateless functions. When offloading, all the objects of the application are maintained locally, and parts of function calls are executed remotely; thus, the application runs normally when the network connection is changing. This section introduces our target program structure that supports computation offloading and its refactoring mechanism (Section IV-B1), mainly including two parts: function wrappers (Section IV-B2) and transmitters (Section IV-B3).

*1) Target Program Structure:* Our target program structure is composed of two elements: function wrapper *Function_T_Wrapper* and function transmitter *Function_T_Transmitter*, as shown in Fig. 4. In this structure, an object is deployed locally, and only its function wrappers are offloaded to a remote server. To enable executing function wrappers on remote servers, we found their external variables via static analysis and modified them to be passed in by parameters and returned by return values. Here, variables external to the function are those accessed within it but declared outside. For example, as shown in Fig. 5, $c$ and $d$ are the external variables of the original function.

The translation to target programs has three steps:

1) Converting function calls from *Function_I* to *Function_T* into indirect calls via *Function_T_Transmitter*.
2) Transforming the inputs and the outputs of *Function_T* to those of *Function_T_Wrapper*. As transformed functions don't access external variables, they are stateless.
3) Generating proxy functions *Function_T_Transmitter* for *Function_T*. *Function_T_Transmitter* has the same function signature as *Function_T*, and it is responsible for determining the execution location of *Function_T_Wrapper*.

After the above translation, *Function_I* calls *Function_T_Transmitter* locally, and *Function_T_Transmitter* decides whether to call *Function_T_Wrapper* locally or remotely according to the offloading decision scheme. The call sites of *Function_I* to *Function_T* are unchanged. We next introduce the process of generating function wrappers (Section IV-B2) and function transmitters (Section IV-B3).

*2) Function Wrapper:* FUNOff generates function wrappers with three steps:

1) Modifying the parameters and return values of a function. As shown in Line 1 of Fig. 5(a) and (b), the parameter $params$ is added to the original function signature, and it records the external variables of the original function. Through this parameter, external variables are passed into the modified function. In addition, the function in Fig. 5(b) is added to return the values of $params$, so the changes on external variables can be returned to function callers.
2) Modifying all statements in the function that access external variables. As shown in Line 2 of Fig. 5(a), $c$ and $d$ are two external variables, and as shown in Line 2 of Fig. 5(b), the two external variables are replaced with $params.c$ and $params.d$. After the modification, the modified function does not access external variables.
3) Modifying all return statements for the function. The return statement of Fig. 5(a) is modified to Lines 3 to 6 of Fig. 5(b). In particular, $params$ is added to the return result, so both the changes in external variables and the return value are returned to function callers.

*3) Function Transmitter:* A function transmitter is the proxy of a function, which is responsible for handling control messages and data synchronization. The construction of the function transmitter includes the following steps:

1) Generating a function whose name, parameters, and return values are identical to the original function, as shown in Line 1 of Fig. 6(b).
2) Adding a statement to record the current function call. FUNOff uses a global variable, called $seq$, to represent the position of the current function call in the call tree. As shown in Line 2 of Fig. 6(b), when calling each a function, FUNOff adds its signature to $seq$ to record the position of a new function call in the call tree, and as shown in Line 15 of Fig. 6(b), it removes the function signature from $seq$ when exiting it.
3) Adding a statement to handle additional variables. As shown in Lines 3 to 5 of Fig. 6(b), a variable of type $Params$ are declared and initialized with the information of local external variables.

```
void function (int a, int b) {
        d = a +b +c;
        return null;
}
```

(a) The source code

```
1:   Void function (int a, int b) {
2:           seq += function.signature;
3:           Params params = new Params();
4:           params.c = c;
5:           params.d = d;
6:           loc = getloc(seq);
7:           Result result = new Result();
8:           if (loc == Local) {
9:                result = function (params, a, b);
10:          } else {
11:               result = remote(seq, params, a, b);
12:          }
13:          c = result.params.c;
14:          d = result.params.d;
15:          seq -= function.signature;
16:          return result.val;
17:  }
```

(b) The target code of function transmitter

Fig. 6. Example of original function and function transmitter, where $c$ and $d$ were external variables (a) The source code. (b) The target code of function transmitter.

4) Adding a statement for the local or remote call to the function wrapper. As shown in Lines 6 to 12 of Fig. 6(b), FUNOff checks the execution location of the current function call in our offloading scheme (Section IV-C1) with $seq$ and calls the local function wrapper or the remote one. In particular, if a function wrapper is called remotely, its parameters and $seq$ are sent to the agent on the remote node, and the remote agent identifies the current function call through $seq$ and invokes it.
5) Adding the statement to receive the result returned by the function wrapper. As shown in Lines 13 to 14 of Fig. 6(b), the local external variables are updated with the result data to ensure the consistency of program states. In addition, the latest return value is returned to its caller, as shown in Line 16 of Fig. 6(b).

### C. Offloading Strategy

In this section, we introduce our offloading strategy. It is designed to minimize the overall offloading cost. We next present the factors that affect the offloading decision (Section IV-C1), the optimization function of our offloading strategy (Section IV-C2), and our offloading decision algorithm to determine the offloading scheme (Section IV-C3).

*1) Contribution Factor:* Offloading schemes determine which functions shall be offloaded and which computing node shall be offloaded. For a given context, it would lead to less overall cost of offloading by using a better offloading scheme. A context contains devices at different scenarios (DS), edge servers (ES), and a cloud server (CS):

*Definition 3.* A context is a graph $G_C = (N, E)$ representing the network environment, where $N$ denotes a set of local devices and remote servers, and $E$ denotes a set of communication links among nodes. Each edge $(n_p, n_q) \in E$ denotes a data transmission whose rate is $v_{n_p - n_q}$ and whose round-trip time $rtt_{n_p - n_q}$ is between $n_p$ and $n_q$.

*Definition 4.* An offloading scheme is defined as $DEP = \{dep(f_1), dep(f_2), \ldots, dep(f_n)\}$, where $f_i$ is a function, and $dep(f_i) \in N$ denotes the computing node to offload the function.

Let $T_{dep(f_i)}^{dep(f_j)}(f_i)$ represents the total offloading time of $f_i$, where $dep(f_i)$ and $dep(f_j)$ denote the offloading positions of $f_i$ and its caller $f_j$. The response time of application be expressed as $T_{response}$, which equals to the sum of $T_{dep(f_i)}^{dep(f_j)}(f_i), f_i \in Tree_{f_{main}}.F$. In addition, $Sinvoke_{n_q}^{f_i} = \langle Stime, Sdatasize \rangle$ is obtained from the estimation model built in AndroidOff [23], where $Stime$ denotes the execution time and $Sdatasize$ denotes the amount of data transmission except external invocations in $f_i$ executed in $n_q$. Note that, $Einvoke$ mentioned in Section IV-A2 is different from $Sinvoke$ in that it contains external invocations shown at the bottom of this page.

*2) Optimization Function:* This section introduces our optimization function (5), and we consider the one with the smallest value as the optimal offloading scheme.

Eq. (5) calculates the response time of $Tree_{f_a}$(the subtree rooted at $f_a$), which consists of the total offloading time $T_{dep(f_i)}^{dep(f_j)}(f_i)$ of all functions in this subtree. When $f_a$ is the $f_{main}$, (5) calculates the response time of the application. Algorithm 4 uses it to calculate the response time.

Eq. (6) calculates the total offloading time of $f_i$, which is composed of the total execution time $T_e{}^{dep(f_i)}(f_i)$ and the total data transmission time $T_{d_{dep(f_i)}}^{dep(f_j)}(f_i)$ of $f_i$ in $dep(f_i)$.

$$\begin{aligned} T_{response}^{f_a} &= T\left(Sinvoke, G_c, Tree_{f_a}, DEP_{f_a}\right) \\ &= \sum_{i=1}^{n} T_{dep(f_j)}^{dep(f_i)}\left(f_i\right), f_i \in Tree_{f_a}.F, \langle f_j, f_i \rangle \in Tree_{f_a}.R.key \end{aligned} \tag{5}$$

$$T_{dep(f_j)}^{dep(f_i)}\left(f_i\right) = T_e{}^{dep(f_i)}\left(f_i\right) + T_{d_{dep(f_j)}}^{dep(f_i)}\left(f_i\right) \tag{6}$$

$$T_e{}^{dep(f_i)}\left(f_i\right) = Sinvoke_{dep(f_i)}^{f_i}.Stime * r_{f_j - f_i} \tag{7}$$

$$T_{d_{dep(f_j)}}^{dep(f_i)}\left(f_i\right) = \left(\frac{Sinvoke_{dep(f_i)}^{f_i}.Sdatasize}{v_{dep(f_j)-dep(f_i)}} + rtt_{dep(f_j)-dep(f_i)}\right) * r_{f_j - f_i}. \tag{8}$$

Eq. (7) calculates the total execution time of $f_i$, which is the product of $Stime$ of $f_i$ in $dep(f_i)$ and its call times.

Eq. (8) calculates the total data transmission time between $f_i$ and $f_j$, which is the sum of the transmission time and the round-trip time. In particular, the transmission time is calculated as the data transmission amount of $f_i$ over the transmission rate of $dep(f_i)$ and $dep(f_j)$.

*3) Offloading Decision Algorithm:* A backtracking algorithm [37] transforms the solution space of a problem into a graph or a tree, which finds the optimal one by enumerating all feasible solutions. Based on the backtracking algorithm [37], we propose an offloading decision algorithm for the call-and-return applications. In a call-and-return application, when a function $A$ calls a function $B$, the result returns to $A$ after $B$ is executed. For each function, our algorithm explores its execution locations by traversing the call tree in the depth-first order. The algorithm calculates the optimization-function value of each scheme and selects the scheme with the minimum value. Meanwhile, the algorithm integrates the depth-first traversal with the following two pruning mechanisms:

1) *Mechanism 1.* A function can be offloaded only if its execution time would be shorter on the offloaded computing node. That is, if a function is executed on the computing node $A$ and its caller function in the call tree is executed on the computing node $B$, the execution time on $A$ must be shorter than that on $B$. Therefore, if there are more available computing nodes, this mechanism tends to reduce more time cost.
2) *Mechanism 2.* When the computing node for the function $f_a$ is determined, the offloading schemes of its subtrees can be decided separately, which are rooted at $f_a$'s callee functions in the call tree. Therefore, this mechanism is able to reduce time cost when a call tree has many branches.

Mechanism 1 can effectively offload functions in most cases. As required by this mechanism, a function can only be offloaded to a remote computing node that outperforms the execution result on the local computing node, because it causes extra data transmission time.

For Mechanism 2, the explanation is given as follows: If a call tree $Tree_{f_a}$ (rooted at $f_a$) contains $n$ subtrees $\{Tree_{f_1}, \ldots, Tree_{f_n}\}$ and $Tree_{f_i}$ is a subtree rooted at function $f_i$, according to (5), the response time of $Tree_{f_a}$ is calculated by (9). When the offloading location of $f_a$, i.e., $dep(f_a)$, is determined, $T^{dep(f_a)}_{dep(f_a.caller)}(f_a)$ is a constant. Meanwhile, $Sinvoke, G_c, Tree_{f_a}, Tree_{f_1}, \ldots, Tree_{f_n}$ are fixed parameters, and $DEP_{f_1}, DEP_{f_2}, \ldots, DEP_{f_n}$ are mutually independent parameters. Thus, the minimum response time of $Tree_{f_a}$ can be calculated by (10), and the offloading schemes of $f_a$'subtrees can be decided separately.

Algorithm 4 describes the decision-making process. For a given call tree Tree $Tree_{f_{main}}$, the algorithm searches for the optimal offloading scheme $DEP_{f_{main}}$ (rooted at $f_{main}$) in a MEC environment $G_c$. Line 1 initially adds a virtual function (denoted by $f_{main}.caller$) to $Tree_{f_{main}}$ and sets the execution locations of $f_{main}$ and $f_{main}.caller$ to the mobile device (DS). Line 2 traverses with the function $getTraversalDEP()$ to obtain $DEP_{f_{main}}$. Lines 3 to 32 define the function $getTraversalDEP()$ that searches for the optimal offloading scheme for the tree or subtree $Tree_{f_{cur}}$ (rooted at $f_{cur}$), which owns the minimum value of optimization function. Line 4 uses $DEP_{best}$ to record the best offloading scheme for $Tree_{f_{cur}}$. Lines 5 to 6 initialize $DEP_{best}$ and calculate its value of optimization function $T_{best}$, in which execution locations of all functions are set to the one of the caller function $f_{cur}.caller$. Line 7 determines whether the computing node for the caller function $f_{cur}.caller$ perform best: If yes, according to Mechanism 1, all functions in $Tree_{f_{cur}}$ should be executed at the same computing node as $f_{cur}.caller$, and then Line 8 returns the initial scheme of $DEP_{best}$ and corresponding $T_{best}$; If no, Lines 9 to 31 search for $DEP_{best}$ by depth-first traversal. Lines 10 to 15 generate candidate computing nodes for executing $f_{cur}$, which are recorded in the set $NodesSet$. Only cut-point functions can be offloaded, and we determine whether $f_{cur}$ is a cut-point function: If no, $NodesSet$ only contains the computing node for the caller function $dep(f_{cur}.caller)$; If yes, $NodesSet$ also contains computing nodes with better performance than

$$
\begin{aligned}
T^{f_a}_{response} &= T\left(Sinvoke, G_c, Tree_{f_a}, DEP_{f_a}\right) \\
&= T^{dep(f_a)}_{dep(f_a.caller)}\left(f_a\right) + T^{f_1}_{response} + T^{f_2}_{response} + \cdots + T^{f_n}_{response} \\
&= T^{dep(f_a)}_{dep(f_a.caller)}\left(f_a\right) + T\left(Sinvoke, G_c, Tree_{f_1}, DEP_{f_1}\right) + T(Sinvoke, G_c, Tree_{f_s}, DEP_{f_s}) \\
&\quad + \cdots + T\left(Sinvoke, G_c, Tree_{f_n}, DEP_{f_n}\right)
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
\min(T^{f_a}_{response}) &= \min(T^{dep(f_a)}_{dep(f_a.caller)}\left(f_a\right) + T^{f_1}_{response} + T^{f_2}_{response} + \cdots + T^{f_n}_{response}) \\
&= \min(T^{dep(f_a)}_{dep(f_a.caller)}\left(f_a\right) + T\left(Sinvoke, G_c, Tree_{f_1}, DEP_{f_1}\right) + T(Sinvoke, G_c, Tree_{f_s}, DEP_{f_s}) \\
&\quad + \cdots + T\left(Sinvoke, G_c, Tree_{f_n}, DEP_{f_n}\right)) \\
&= \min(T^{dep(f_a)}_{dep(f_a.caller)}\left(f_a\right)) + \min(T\left(Sinvoke, G_c, Tree_{f_1}, DEP_{f_1}\right)) + \min(T(Sinvoke, G_c, Tree_{f_s}, DEP_{f_s})) \\
&\quad + \cdots + \min(T\left(Sinvoke, G_c, Tree_{f_n}, DEP_{f_n}\right)).
\end{aligned} \tag{10}
$$

**Algorithm 4:** Offloading Decision Algorithm.

**Input:** A call tree $Tree_{f_{main}} = (F, R)$; a context environment $G_c = (N, E)$; a set of execution costs for each function except external invocations $Sinvoke$

**Output:** An offloading scheme $DEP_{f_{main}} = \{dep(f_1), dep(f_2), \ldots, dep(f_n)\}$; the response time $T_{response}$

1: $DEP_{f_{main}}.dep(f_{main}), DEP_{f_{main}}.dep$ $(f_{main}.caller) \leftarrow DS$
2: $DEP_{f_{main}}, T_{response} \leftarrow getTraversalDEP(f_{main}, DEP_{f_{main}}.dep(f_{main}.caller))$
3: **function** getTraversalDEP$f_{cur}, dep(f_{cur}.caller)$
4: $\quad DEP_{best} \leftarrow DEP_{f_{cur}}$
5: $\quad DEP_{best}.dep(f_i) \leftarrow dep(f_{cur}.caller), \forall f_i \in Tree_{best}$
6: $\quad T_{best} \leftarrow$ optimization function$(Sinvoke, G_c, Tree_{f_{cur}}, DEP_{best})$
7: $\quad$ **if** $dep(f_{cur}.caller)$ is the best performing computing node **then**
8: $\quad\quad$ **return**$DEP_{best}, T_{best}$
9: $\quad$ **else**
10: $\quad\quad NodesSet \leftarrow \emptyset$, $NodesSet \leftarrow NodesSet + dep(f_{cur}.caller)$
11: $\quad\quad$ **if** $f_{cur} \in divFunction$ **then**
12: $\quad\quad\quad$ **for** each computing node $n$ with better performance than $dep(f_{cur}.caller)$ **do**
13: $\quad\quad\quad\quad NodesSet \leftarrow NodesSet + n$
14: $\quad\quad\quad$ **end for**
15: $\quad\quad$ **end if**
16: $\quad\quad$ **for** each $n \in NodesSet$ **do**
17: $\quad\quad\quad DEP_{temp} \leftarrow DEP_{f_{cur}}$
18: $\quad\quad\quad DEP_{temp}.dep(f_{cur}) \leftarrow n$
19: $\quad\quad\quad$ **if** $post(f_{cur}) \neq \emptyset$ **then**
20: $\quad\quad\quad\quad$ **for** each $f_i$ in $post(f_{cur})$ **do**
21: $\quad\quad\quad\quad\quad DEP, T \leftarrow getTraversalDEP(f_i, n)$
22: $\quad\quad\quad\quad\quad DEP_{temp}.dep(f_j) \leftarrow DEP.dep(f_j), \forall f_j \in Tree_{f_i}$
23: $\quad\quad\quad\quad$ **end for**
24: $\quad\quad\quad$ **end if**
25: $\quad\quad\quad T_{temp} \leftarrow$ optimization function$(Sinvoke, G_c, Tree_{f_{cur}}, DEP_{temp})$
26: $\quad\quad\quad$ **if** $T_{temp} < T_{best}$ **then**
27: $\quad\quad\quad\quad T_{best} \leftarrow T_{temp}$, $DEP_{best} \leftarrow DEP_{temp}$
28: $\quad\quad\quad$ **end if**
29: $\quad\quad$ **end for**
30: $\quad\quad$ **return**$DEP_{best}, T_{best}$
31: $\quad$ **end if**
32: **end Function**

$dep(f_{cur}.caller)$, according to Mechanism 1. Lines 16 to 29 respectively perform a depth-first traversal of $Tree_{f_{cur}}$, for each candidate computing node $n$ for $f_{cur}$ ($n \in NodesSet$). According to Mechanism 2, when the execution location of $f_{cur}$ is fixed, the offloading schemes of its subtrees can be decided separately, which are rooted at $f_{cur}$'s callee functions in the call tree. The traversal is as follows: Lines 17 to 18 use $DEP_{temp}$ to record the best offloading scheme for $Tree_{f_{cur}}$ when the execution location of $f_{cur}$ is $n$. To obtain $DEP_{temp}$, Lines 19 to 24 call the function $getTraversalDEP()$ for each $f_{cur}$'s calee functions $f_i$ to obtain the best offloading scheme for $Tree_{f_i}$. Lines 25 to 28 calculate the optimization-function value of $DEP_{temp}$, and update $DEP_{best}$ if it is less than the current $DEP_{best}$. When the traversal (Lines 16 to 29) is completed, Line 30 returns $DEP_{best}$ and the corresponding $T_{best}$. Based on the function $getTraversalDEP()$, the optimal offloading scheme $DEP_{f_{main}}$ can be obtained.

## V. Evaluation

In this section, we established an MEC environment to evaluate the effectiveness of FUNOff (Section V-A). In this environment, we compared FUNOff with AndroidOff [23], [24] and MAUI [9] (Section V-B). Beside the overall effectiveness, we conducted experiments to explore the details of FUNOff (Section V-C).

### A. MEC Environment

Our MEC environment includes two scenes (college and community), and each scene contains four regular locations. In total, our experimental environment uses five computing nodes, including two mobile devices and three remote servers. Table III lists the network conditions between these computing nodes, where each cell denotes the round-trip time and the data transmission rate between our mobile devices and corresponding

TABLE III
THE DEVICE CONTEXTS

(a) College

| | Playground | Teaching building | Garden | Laboratory | Cloud |
|---|---|---|---|---|---|
| Edge1 | - | RTT = 40$ms$ V = 1.5$Mb/s$ | RTT = 40$ms$ V = 1.5$Mb/s$ | - | RTT = 40$ms$ V = 1.5$Mb/s$ |
| Edge2 | - | - | RTT = 70$ms$ V = 1.0$Mb/s$ | RTT = 40$ms$ V = 1.5$Mb/s$ | RTT = 40$ms$ V = 1.5$Mb/s$ |
| Cloud | RTT = 200$ms$ V = 200$kb/s$ | RTT = 200$ms$ V = 200$kb/s$ | - | RTT = 70$ms$ V = 1.0$Mb/s$ | - |

(b) Community

| | Residence | Traffic Road | Parking Lot | Store | Cloud |
|---|---|---|---|---|---|
| Edge1 | - | RTT = 40$ms$ V = 1.5$Mb/s$ | - | RTT = 60$ms$ V = 1.2$Mb/s$ | RTT = 20$ms$ V = 2$Mb/s$ |
| Edge2 | RTT = 60$ms$ V = 1.2$Mb/s$ | RTT = 70$ms$ V = 1.0$Mb/s$ | - | - | RTT = 20$ms$ V = 2$Mb/s$ |
| Cloud | RTT = 100$ms$ V = 500$kb/s$ | RTT = 100$ms$ V = 500$kb/s$ | RTT = 100$ms$ V = 500$kb/s$ | RTT = 100$ms$ V = 500$kb/s$ | - |

TABLE IV
THE PERFORMANCE OF COMPUTING NODES

| | Device1 | Device2 | Edge1 | Edge2 | Cloud |
|---|---|---|---|---|---|
| Performance evaluation | 1 | 1.2 | 2.2 | 2.8 | 4.4 |

remote servers. For example, in Table III(a), the fourth column of the second row denotes that the round-trip time between our mobile device and Edge1 is 40$ms$, and the transmission rate reaches 1.5Mb/s in the garden. The data in Table III are collected by WLAN-RTT.[2]

We have installed the applications on two mobile devices. One mobile device is Huawei Honor MYA-AL10[3] with a 1.4 GHz 4 core CPU, 2 GB RAM (Device1, the low-end device) and the other is Huawei Honor STF-AL00[4] with 2.4 GHz 4 core CPU, 4 GB RAM (Device2, the high-end device). Our MEC environment has two edge servers (Edge1 and Edge2) and a cloud server (cloud). Edge1 is a server with a 2.5 GHz 8 core CPU and 4 GB RAM; Edge2 is a server with a 3.0 GHz 8 core CPU and 8 GB RAM; Cloud is a server with a 3.6 GHz 16 core CPU and 16 GB RAM. To measure the performance of each computing node, we execute an identical set of functions, and compare the execution time with that on Device1. Table IV shows the results.

In our evaluations, the subject applications include a License Plate Recognition Application (LPRA) and a Target Detection Application (TDA). LPRA performs preprocessing and ocr processing on the images that are extracted from video frames to obtain the license plate numbers, and stores them on the mobile device. TDA performs pedestrian detection and feature extraction on the images extracted from the video and saves the results on the mobile device after feature comparison with the person to be recognized. We installed them on both mobile devices. In our experiments, we walk around the above two scenes and execute these two applications. In this process, we record the data transmission amount and the execution time of each function call on devices, Edge1, Edge2, and Cloud. Upholding the principle of rigor, we repeat this process twenty times to avoid unnecessary errors. For example, Fig. 7(a) shows the collected LPRA data on the Huawei Honor MYA-AL10. The ellipse indicates the function, and the data above it indicates the execution time of the function on this device. For example, 16 in the dashed box indicates the time (in *ms*) of one execution of the function OAlg.gm() on Huawei Honor MYA-AL10. The connecting line indicates the call relationship between the functions, and its data indicates the number of calls and the amount of data transferred between them. For example, 1:280 in the dashed box indicates the function OAlg.Graymath() makes one call to the function OAlg.gm(), and the amount of data transfer generated by one call is 280B. Fig. 7(b) shows the collected TDA data on the Huawei Honor MYA-AL10.

The parameters $\lambda$, $v$ and $rtt$ used in the preprocessing algorithm (Algorithm 3) need to be set according to the ideal offloading environment. To find all possible cut-points during the preprocessing phase, the ideal offloading environment in our experimental environments (i.e., from the Huawei Honor MYA-AL10 to the Edge2 in the laboratory of college) is selected with the consideration of server performance and data transmission rate to conduct the simulation offloading experiment of Algorithm 3. $\lambda$ is set to 2.8 based on the performance ratio between MYA-AL10 and Edge2, as shown in Table IV. $v$ and $rtt$ are set to 1.5 *Mb/s* and 40$ms$, respectively, based on the network connection between them, as shown in Table III(a).

## B. Overall Comparison

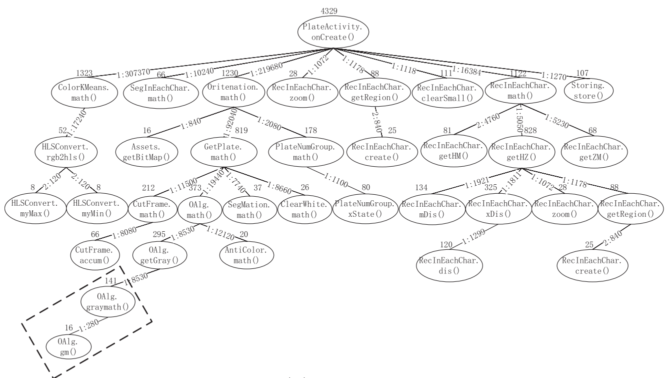
*1) Compared Approach and Scenarios:* In this section, we compared FUNOff with AndroidOff [23], [24] and MAUI [9]. AndroidOff works at the granularity of objects. It traverses all possible deployments from the mobile device to servers, and searches for the decision that can minimize the response time. MAUI works at the granularity of methods. It uses integer linear programming to decide where the movable functions shall be moved to servers.

Owing to the mobility of devices, we considered the following two scenarios: (1) we stay in different fixed locations with mobile devices (Section V-B2) and (2) with mobile devices, we move between different locations in the college and community respectively (Section V-B3). We use the response time generated by the real execution of the application as the metric of performance. In addition to task execution and data transmission time, the response time includes the additional time overhead generated by the mechanisms. Each experiment is repeated for 20 times to ensure its reliability [22].

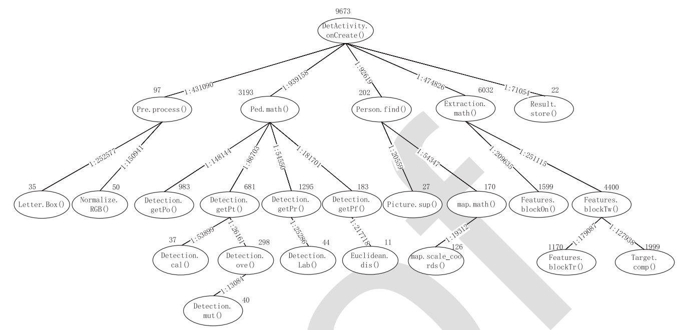
*2) Performance Comparison of Fixed Locations:* Fig. 8 shows that FUNOff achieves the best performance in all cases. Fig. 9 shows the offloading schemes of AndroidOff, MAUI, and FUNOff when running LPRA on Honor MYA-AL10 in the garden.

Comparing the functions of the RecInEachChar class in Fig. 9(a) with (b) we find that FUNOff offloaded the instances of these functions to three computing nodes (Edge2, Cloud, and Device1). Note that the device can connect to the Cloud via Edge1 or Edge2; AndroidOff offloaded the instances of

[2]https://developer.android.google.cn/guide/topics/connectivity/wifi-rtt
[3]http://huawei-update.com/device-list/yma-al10
[4]http://huawei-update.com/device-list/stf-al00

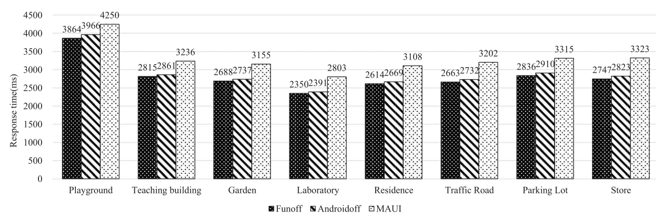


(a) LPRA

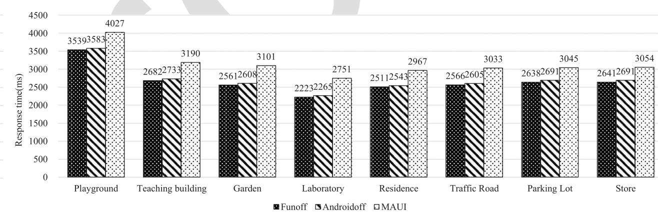


(b) TDA

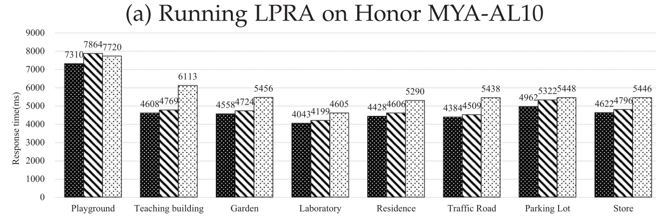
Fig. 7. LPRA and TDA performed on Huawei Honor MYA-AL10 (a) LPRA. (b) TDA.



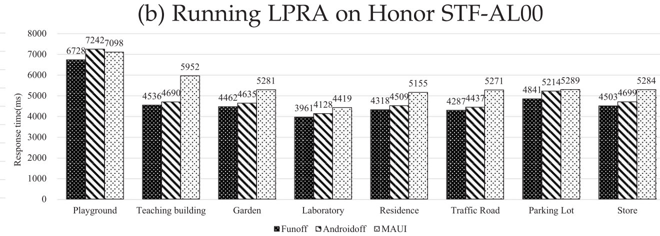

(a) Running LPRA on Honor MYA-AL10




(b) Running LPRA on Honor STF-AL00




(c) Running TDA on Honor MYA-AL10




(d) Running TDA on Honor STF-AL00

Fig. 8. Performance comparison of running LPRA and TDA with different offloading approaches when staying in different locations fixedly (a) Running LPRA on Honor MYA-AL10. (b) Running LPRA on Honor STF-AL00 (c) Running TDA on Honor MYA-AL10 (d) Running TDA on Honor STF-AL00.

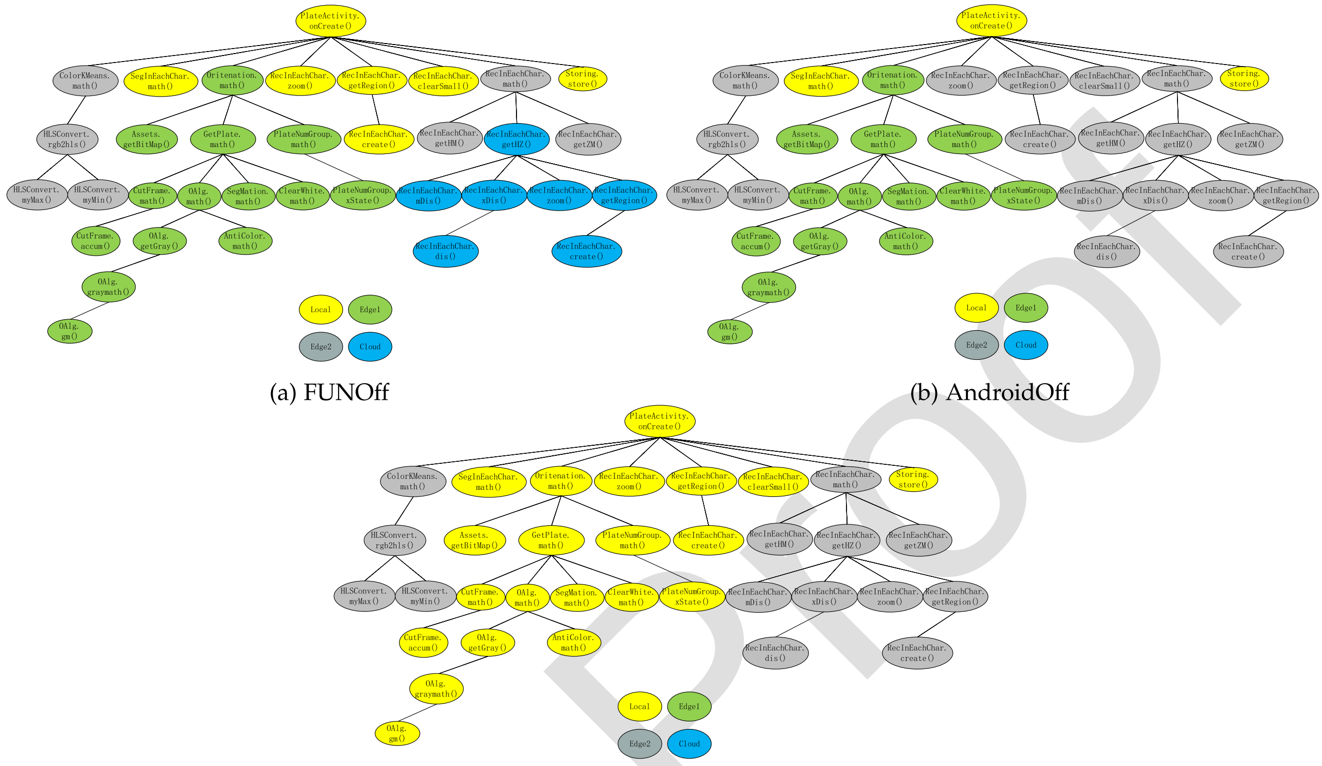
the whole class to Edge2. As our offloading granularity is finer, FUNOff is more flexible than AndroidOff. As a result, it improves the results of AndroidOff.

Comparing Fig. 9(a) with (c), we find that MAUI moved all methods to a single server, and this scheme is sub-optimized. Instead, as our offloading decision can weigh the different network connections, FUNOff offloaded the functions whose data transmission is intensive to remote servers with good network connections. Meanwhile, as our offloading decision can weigh the different performance of servers, FUNOff offloaded the functions whose computation is intensive to remote servers with better computation power but relatively poor network connections.

To further analyze our improvements, we next introduce the results of LPRA, when it is installed on Honor MYA-AL10 and moved around the playground. Both FUNOff and MAUI support offloading at function granularity, and only a cloud server is available here, so their offloading schemes are the same. However, the results in Fig. 8(a) show that FUNOff still improves by about 10% over MAUI. This is because the offloading mechanism introduces additional overhead such as the execution of extra statements, the response time of the server, etc. Since FUNOff only refactors the cut-point functions, while MAUI needs to refactor all the methods, this causes more additional overhead. And AndroidOff will incur an overhead of approximately 170 *ms*, which originates from the proxies.

*3) Performance Comparison When Cruising Between Different Locations:* Due to the different computing resources and network connections in locations, the offloading scheme needs to be updated when a mobile device moves to a new location. The results from Honor MYA-AL10 and Honor STF-AL00 in both the college and community scenes are consistent. For simplicity, we only show the results of MYA-AL10 when it is in the college. Fig. 10 shows the decision and preparation costs in the four locations of the college scene. According to the results, FUNOff has the following advantages:

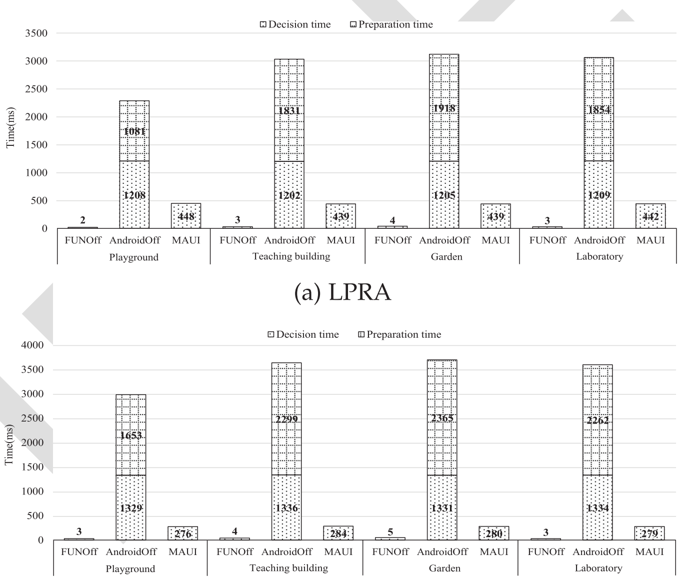
(1) FUNOff has the least decision time. For this measure, the averages of FUNOff, AndroidOff, and MAUI on LPRA are 218*ms*, 1,206*ms*, and 442*ms*, and the averages on TDA are 3.8*ms*, 1333*ms*, and 280*ms*, respectively. FUNOff only decides the offloading position of cut-point functions, and different branches can make decisions independently, the details of Algorithm 3 are




(a) FUNOff (b) AndroidOff

(c) MAUI

Fig. 9. Offloading schemes when running LPRA on the Honor MYA-AL10 in garden (a) FUNOff. (b) AndroidOff (c) MAUI.




(a) LPRA

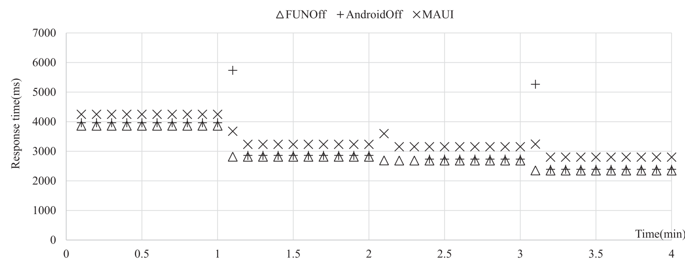
(b) TDA

Fig. 10. The time cost of adjusting offloading schemes of different offloading approaches for LPRA and TDA on the Honor MYA-AL10 in the college (a) LPRA. (b) TDA.

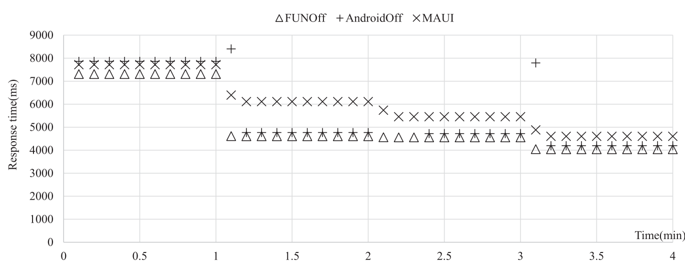
shown in Section IV-C3. Therefore, it can make decisions in a short time. AndroidOff is based on traversal and needs to select the best one from all possible object distribution schemes. Therefore, its decision time is exponentially related to the number of movable objects. MAUI is based on the program partitioning strategy, and determines offloading schemes at runtime. Therefore, its decision time is linearly related to the number of movable methods. The compared approaches require more decision times than FUNOff.

(2) When the network connection changes, FUNOff and MAUI do not need extra preparations for the new compute offloading, but the average preparation time of AndroidOff on LPRA and TDA are 1,671*ms* and 2145*ms*, respectively. Both FUNOff and MAUI offload applications at the granularity of functions (methods), and they store program states on mobile devices. As a result, functions can be executed directly on a new remote server when the network connection changes. AndroidOff offloads applications at the granularity of objects, and objects are executed on either mobile devices, edge servers, or cloud servers. When an offloading scheme changes, the application needs to offload the objects from an old computing node to a new computing node. Moreover, if an offloaded object becomes inaccessible, the application crashes and has to be restarted.

Fig. 11 shows results on Honor MYA-AL10. FUNOff has the best results; AndroidOff is the second in most cases; and MAUI has the worst. FUNOff and AndroidOff can use multiple remote servers for computation offloading, but MAUI is designed to use a single remote server. When the device context changes, the response time of FUNOff and MAUI only increases slightly due to the additional cost caused by making decisions. In contrast, the response time of AndroidOff increases by about three seconds, mainly due to the decision time and the offloading

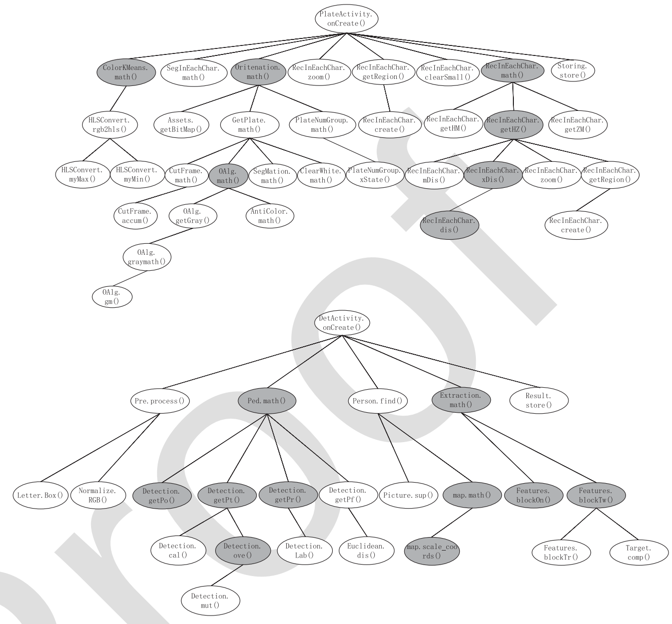



(a) LPRA

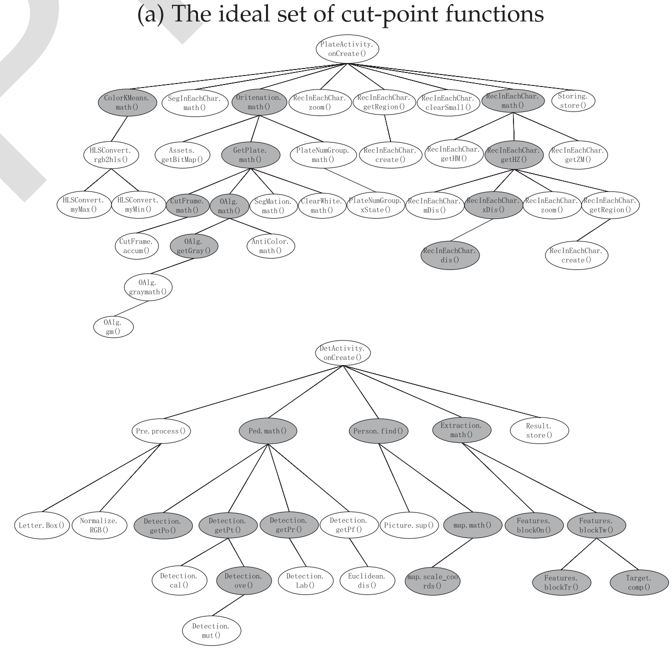



(b) TDA

Fig. 11. Performance comparison of running LPRA and TDA on Honor MYA-AL10 with different offloading approaches when cruising between four locations in the college (a) LPRA. (b) TDA.



(a) The ideal set of cut-point functions



(b) The cut-point set obtained by Algorithm 3

Fig. 12. Sets of the cut-point functions of LPRA and TDA (a) The ideal set of cut-point functions. (b) The cut-point set obtained by Algorithm 3.

preparation time. In addition, when the device cruised from the teaching building to the garden, AndroidOff failed to respond for about twenty seconds. The original object on the cloud was inaccessible, so the application crashed and restarted.

### C. Detailed Comparison

In this section, we explore the effectiveness of the cut-point algorithm (Section V-C1), the offloading schemes of different decision algorithms (Section V-C2), and their time costs (Section V-C3).

*1) Evaluation of the Cut-Point Algorithm: Setting.* In this section, we evaluate the rationality and feasibility of the cut-point algorithm (Algorithm 3) which extracts cut-point functions from the call tree.

For each case, we analyze the call tree and the MEC environment to manually obtain the corresponding optimal offloading scheme. Next, we take the union of functions offloaded in those offloading schemes as the ideal set of cut-point functions, i.e., the gray nodes as shown in Fig. 12(a). We compare the set obtained by the cut-point algorithm (Algorithm 3) with the ideal set. If the cut-point set covers the ideal set, our cut-point algorithm can find all the functions offloaded in those optimal offloading schemes and will not affect the search for the optimal offloading scheme. If the cut-point set contains redundant cut-point functions, the extra number of decisions due to the extra cut-point functions will incur additional decision overhead. With the parameters set in Section V-A, we use Algorithm 3 to calculate the cut-point set, and compare it with the ideal set.

*Result.* Fig. 12(b) shows the results of the cut-point algorithm. Comparing this figure with Fig. 12(a), the ideal set can be covered by the cut-point set obtained by the cut-point algorithm. Meanwhile, the additional decision cost caused by the redundant cut-point functions in our set is acceptable, which will be discussed in Section V-C3.

*2) Evaluation of the Offloading Decision Algorithm: Setting.* In this section, we compare our decision algorithms with the traversal algorithm [23], [24], the Q-learning [38], the particle swarm optimization with the genetic algorithm (PSO-GA) [31], and the classical genetic algorithm (GA) [30]. In particular, our comparison includes two stages: with or without our preprocessing step, which extracts cut-point functions.

*Traversal Algorithm.* The unpreprocessed traversal algorithm obtains the optimal offloading scheme by enumerating the combinations of all the functions on different computing nodes. The

TABLE V
RESULTS OF OFFLOADING SCHEMES OBTAINED BY DIFFERENT DECISION ALGORITHMS

(a) LPRA

| Device and location / Algorithm (performance gap with optimal scheme (%)) | | | Optimal scheme (Traversal Algorithm Without preprocessing) | Traversal Algorithm | Our Algorithm | | Q-learning | | PSO-GA | | GA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing |
| Honor MYA-AL10 | College | Playground | **3864.4ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 2.947% | 0.531% | 5.509% | 2.531% |
| | | Teaching building | **2814.8ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 11.142% | 0.508% | 21.711% | 2.008% |
| | | Garden | **2687.7ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 8.896% | 0.484% | 15.683% | 1.838% |
| | | Laboratory | **2349.8ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 8.669% | 0.813% | 13.473% | 2.072% |
| | Community | Residence | **2613.9ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 8.195% | 0.318% | 10.035% | 1.546% |
| | | Traffic Road | **2662.9ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 5.310% | 0.109% | 9.001% | 0.792% |
| | | Parking Lot | **2836.3ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 5.014% | 0.328% | 8.254% | 0.709% |
| | | Store | **2747.4ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 7.360% | 0.335% | 13.340% | 1.350% |
| Honor STF-AL00 | College | Playground | **3539.1ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 1.138% | 0.330% | 1.314% | 0.554% |
| | | Teaching building | **2682.4ms** | ✓ | ✓ | ✓ | 1.789% | ✓ | 11.014% | 0.933% | 18.574% | 2.003% |
| | | Garden | **2560.8ms** | ✓ | ✓ | ✓ | 4.374% | ✓ | 8.218% | 0.575% | 13.892% | 1.556% |
| | | Laboratory | **2223.2ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 8.940% | 0.817% | 13.159% | 1.928% |
| | Community | Residence | **2510.7ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 9.053% | 0.315% | 13.941% | 1.386% |
| | | Traffic Road | **2565.7ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 6.945% | 0.226% | 15.208% | 0.760% |
| | | Parking Lot | **2638.3ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 4.219% | 0.023% | 12.910% | 0.788% |
| | | Store | **2640.6ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 7.726% | 0.360% | 14.364% | 1.015% |

(b) TDA

| Device and location / Algorithm (performance gap with optimal scheme (%)) | | | Optimal scheme (Traversal Algorithm Without preprocessing) | Traversal Algorithm | Our Algorithm | | Q-learning | | PSO-GA | | GA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing | Without preprocessing | With preprocessing |
| Honor MYA-AL10 | College | Playground | **7309.7ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 13.325% | 1.316% | 17.250% | 5.241% |
| | | Teaching building | **4608.1ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 26.454% | 4.681% | 39.778% | 11.094% |
| | | Garden | **4557.5ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 28.024% | 3.291% | 41.068% | 9.521% |
| | | Laboratory | **4042.7ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 32.355% | 6.258% | 45.972% | 15.312% |
| | Community | Residence | **4428.3ms** | ✓ | ✓ | ✓ | 1.287% | ✓ | 27.080% | 7.023% | 39.021% | 17.094% |
| | | Traffic Road | **4384.4ms** | ✓ | ✓ | ✓ | 0.319% | 0.068% | 18.943% | 7.960% | 33.396% | 13.274% |
| | | Parking Lot | **4962.0ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 25.058% | 8.102% | 31.929% | 14.833% |
| | | Store | **4622.5ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 27.636% | 7.745% | 45.773% | 17.977% |
| Honor STF-AL00 | College | Playground | **6727.9ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 12.481% | 6.416% | 15.552% | 10.826% |
| | | Teaching building | **4536.4ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 23.343% | 1.869% | 36.925% | 11.502% |
| | | Garden | **4462.4ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 24.843% | 1.791% | 37.599% | 8.815% |
| | | Laboratory | **3961.1ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 24.135% | 0.444% | 39.863% | 7.885% |
| | Community | Residence | **4317.5ms** | ✓ | ✓ | ✓ | 3.868% | ✓ | 26.201% | 7.736% | 40.173% | 17.232% |
| | | Traffic Road | **4286.7ms** | ✓ | ✓ | ✓ | 1.656% | 1.353% | 20.065% | 5.692% | 33.936% | 13.390% |
| | | Parking Lot | **4841.5ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 17.242% | 7.539% | 25.426% | 11.402% |
| | | Store | **4502.5ms** | ✓ | ✓ | ✓ | ✓ | ✓ | 24.566% | 8.262% | 41.624% | 18.301% |

preprocessed traversal algorithm enumerates only the cut-point functions.

*Our Algorithms.* We design two versions of Algorithm 4. The original version is called the preprocessed decision algorithm, which makes decisions for the cut-point functions. Another version is called the unpreprocessed decision algorithm, which makes decisions on the execution location for all functions of the call tree.

*Q-Learning.* It stores each state-action pair and its corresponding Q-values into a Q-table, and maximizes the accumulative rewards of an offloading plan. The learning rate $\alpha$, the discount factor $\beta$, the probability of $\varepsilon$-greedy, and the max training epochs are set to 0.01, 0.95, 0.1, and 100,000, respectively. The algorithm will terminate and return the best one when the result is constant for 5,000 consecutive iterations. The unpreprocessed Q-learning needs to make decisions for all functions, while the preprocessed Q-learning only makes decisions for cut-point functions.

*PSO-GA.* It introduces the crossover and mutation operators of GA to improve the particle update strategy of the traditional PSO algorithm. The unpreprocessed version encodes all the functions into a chromosome, and the preprocessed version only encodes the cut-point functions. The start and end values of the two acceleration coefficients $c_1$ and $c_2$, and the maximum and minimum values of the inertia weight $w$ are set to 0.9, 0.2, 0.9, 0.4, 0.9, and 0.4, respectively. The iteration number and population number of the unpreprocessed PSO-GA are set to 2000 and 150, while the preprocessed ones are set to 1100 and 80.

*GA.* The unpreprocessed genetic algorithm encodes all the functions into a chromosome, applies genetic operations (e.g., selection, crossover, and mutation) to generate new offloading schemes, and uses the optimization function to select the best ones. The evolutionary generation, the population number, the crossover probability, and the mutation probability are set as 2,000, 150, 0.6, and 0.3. The preprocessed genetic algorithm only encodes the cut-point functions, and its parameters are set as 1,100, 80, 0.6, and 0.3, respectively.

As the traversal algorithm enumerates all candidate offloading schemes, it is able to find the optimal scheme. We take its optimal scheme and response time as the baseline. If the response time corresponding to the offloading scheme obtained by other algorithms is consistent with it, it means that they find the optimal scheme. If the response time is larger than the baseline, the algorithm finds an offloading scheme with a worse performance than the optimal scheme, and the larger the response time, the worse the performance. Each algorithm is repeated 20 times separately and the average value is taken as its final result.

*Result.* The experimental results are shown in Table V. The tick in this table indicates that the corresponding algorithm finds the optimal offloading scheme. The gray part indicates that it does, and the values denote the increased response time compared with the optimal offloading scheme. For example, in the scenario of Honor MYA-AL10 in the playground in Table V(a), the response time corresponding to the optimal scheme is 3864.4*ms*. In this scenario, the response time corresponding to the offloading scheme obtained from the unpreprocessed PSO-GA is 3978.3*ms*, which is an increase of 2.947% compared to 3864.4*ms*, so the value of the corresponding position in Table V(a) is set to 2.947%.

First, we compare the performance of the algorithms without preprocessing. As shown in Table V, our algorithm achieves the same performance as the traversal algorithm for all the 32 cases. Our algorithm finds the optimal schemes, since it is an improved traversal algorithm and its two effective pruning mechanisms are unlikely to affect the search for the optimal offloading scheme (Section IV-C3 for more details). The Q-learning adaptively learns appropriate scheduling decisions by interacting with the network environment and can obtain the same results as the traversal algorithm in 26 of 32 total cases. However, in other 6 cases, its response time is 0.319%-4.374% higher than the traversal algorithm. Unlike the traversal algorithm that enumerates all candidate offloading schemes, the learning process of Q-learning is uncertain. As the low occurrence of some states causes the randomness of the Q-table, Q-learning is unable to achieve an optimal offloading scheme in some cases. PSO-GA cannot obtain the optimal offloading schemes in all cases, and its response time is 1.138%-32.355% more than the optimal offloading scheme. Although PSO-GA improves the stochasticity through the crossover operations, it still suffers from local optimums. Therefore, PSO-GA fails to obtain the global optimal scheme in a large solution space. Similarly, GA cannot obtain the optimal offloading schemes in all cases, and its response time is 1.314%-45.972% more than the optimal offloading scheme. GA has strong stochasticity and converges slowly, and thus it is difficult to converge to a better offloading scheme with a limited number of iterations.

Furthermore, we compare the performance of each algorithm with and without preprocessing. The traversal algorithm with preprocessing still obtains the optimal scheme in all cases, because the cut-point set obtained by the cut-point algorithm (Algorithm 3) can cover the ideal set, as analyzed in Section V-C1. Similarly, our algorithm with preprocessing can obtain the same scheme without processing in each case. The Q-learning with preprocessing can find the optimal scheme in more scenarios than the one without preprocessing, indicating that our preprocessing algorithm can improve the performance of Q-learning by reducing the size of the solution space, and thus enhance the probability of finding a better state. For PSO-GA and GA, the performance is significantly improved in all cases with preprocessing, although the optimal solution cannot be obtained. For PSO-GA with processing, the response time of its offloading scheme is reduced by 0.8%-19.7% compared to that without processing. For GA with processing, the response time of its offloading scheme is reduced by 0.7%-22.9% compared to that without processing. As the algorithm with processing only makes decisions on the cut-point functions, it drastically reduces the size of the solution space, allowing the algorithms to find better offloading schemes more efficiently.

TABLE VI
COMPARISON OF DECISION TIME

(a) LPRA

| Decision time(ms) / Algorithm / Preprocessing | Traversal algorithm | Our algorithm | Q-learning | PSO-GA | GA |
|---|---|---|---|---|---|
| Without preprocessing | 313884 | 8 | 431 | 3173 | 2138 |
| With preprocessing | 1695 | **3** | 47 | 386 | 305 |

(b) TDA

| Decision time(ms) / Algorithm / Preprocessing | Traversal algorithm | Our algorithm | Q-learning | PSO-GA | GA |
|---|---|---|---|---|---|
| Without preprocessing | 232985 | 7 | 579 | 2616 | 1842 |
| With preprocessing | 4159 | **4** | 76 | 452 | 337 |

*3) The Time Cost of Decision Algorithm: Setting.* The experimental setup is the same as Section V-C2, but we record the decision time to explore their cost.

*Result.* As shown in Table VI, compared to other algorithms, the average decision time of our algorithm is the shortest on both LPRA and TDA. On LPRA, the decision time of our unpreprocessed algorithm is 8*ms*, which saves 98.1%-99.9% compared to other unpreprocessed algorithms. Moreover, the decision time of our preprocessed algorithm is 3*ms*, which saves 93.6%-99.9% compared to other preprocessed algorithms. On TDA, the decision time of our unpreprocessed algorithm is 7*ms*, which saves 98.8%-99.9% compared to other unpreprocessed algorithms. Moreover, the decision time of our preprocessed algorithm is 4*ms*, which saves 94.7%-99.9% compared to other preprocessed algorithms.

For both preprocessed and unpreprocessed algorithms, our algorithm, Q-learning, PSO-GA, and GA reduced the costs of the traversal algorithm by 99.5%, 62.5%, 89.1%, 87.8% and 85.7% on LPRA, and 98.2%, 42.6%, 86.9%, 82.7% and 81.7% on TDA, respectively. Our preprocessing step extracts cut-point functions, reducing the search space and decision times (Section V-C1 for more details). As a result, our preprocessing effectively improves the performance of all decision algorithms.

## VI. Discussion

### A. Extending to Other Applications

Our work focuses on object-oriented applications in Java. Our algorithm is mainly designed for the call-and-return applications, and it needs to be extended to other styles of applications (e.g., workflow applications and DNN-based applications). For example, in a workflow application, a function $B$ is called by function $A$, but passes its execution result to a function $C$. On one hand, the offloading mechanism proposed in this paper can be extended to different types of applications. To support the applications offloading at function granularity in MEC, the statelessness of functions is of utmost importance, since it needs to avoid the loss of state information when the environment changes. For example, each neural network layer of the DNN model can be considered as a stateless function, since all parameters required for the computation of each layer are directly passed in through the input. This style is simpler than OO applications because it does not require any additional transformation. On the other hand, the cut-point function extraction (Algorithm 3) can be extended to other types of applications to reduce the decision overhead. For example, fully connected layers in DNN models, which usually have high execution latency, are suitable to be offloaded and can be considered as cut-points. And neural networks with low execution latency and high data transmission, such as activation layers, are more suitable to be executed on the same computing node as their preceding layers.

### B. Evaluating in Real-World Environments

In our evaluations, we established an MEC environment to maximize the simulation of the real-world environment. The two mobile devices represent low-performance and high-performance devices, and the network conditions between the mobile devices and the remote servers vary by locations. The results reveal the effectiveness of our approach. The differences between our MEC environment and the real-world environments are that: (1) the application runs in a single-user environment. Therefore, the execution time of each call to the same class of methods on the same computing node is generally close to their average; (2) Our mobility model for mobile devices is simplified. We ignore the wireless fading channel caused by device movements, so the network conditions between a mobile device and the same remote server in the same location are generally close to their average. Despite the above differences, our approach can still work in the real-world environment, just with some performance difference. In addition, this study focuses primarily on supporting the dynamic offloading of applications in MEC at function granularity; the two issues above are orthogonal to the problem in this study. In future work, we will consider the above factors, such as supporting multi-user cases via game-theoretic models [39], [40] and supporting complex mobility models through other offloading decision algorithms [32], [41].

## VII. Conclusion

To make use of the scattered and changing computing resources in MEC, this paper proposes an adaptive offloading approach, called FUNOff, which supports the offloading at the granularity of functions. For an object-oriented application, it extracts a call tree through code analysis, and takes a preprocessing step to find the function invocations suitable for offloading. Next, FUNOff translates such functions to a specific program structure that allows remote access. Finally, it generates an offloading scheme at runtime according to the context of the mobile device, and sends functions to multiple devices according to the offloading scheme. Our evaluations on real applications show that FUNOff significantly improves the performance of applications. In addition, the results show that the offloading at the granularity of functions is more suitable for computation offloading in MEC, and our preprocessing effectively improves the performance of offloading decision algorithms.

## References


[1] Z. Su, Y. Hui, and T. H. Luan, "Distributed task allocation to enable collaborative autonomous driving with network softwarization," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2175–2189, Oct. 2018.

[2] C. Liu et al., "A new deep learning-based food recognition system for dietary assessment on an edge computing service infrastructure," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 249–261, Mar./Apr. 2018.

[3] T. Braud, F. H. Bijarbooneh, D. Chatzopoulos, and P. Hui, "Future networking challenges: The case of mobile augmented reality," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 1796–1807.

[4] T. Zhao, J. Liu, Y. Wang, H. Liu, and Y. Chen, "Towards low-cost sign language gesture recognition leveraging wearables," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 1685–1701, Apr. 2021.

[5] F. Yang, J. Li, T. Lei, and S. Wang, "Architecture and key technologies for Internet of Vehicles: A survey," *IEEE J. Commun. Inf. Netw.*, vol. 2, no. 2, pp. 1–17, 2017.

[6] S. Jeong, O. Simeone, and J. Kang, "Mobile edge computing via a UAV-mounted cloudlet: Optimization of bit allocation and path planning," *IEEE Trans. Veh. Technol.*, vol. 67, no. 3, pp. 2049–2063, Mar. 2018.

[7] I. A. Elgendy, W. Zhang, Y.-C. Tian, and K. Li, "Resource allocation and computation offloading with data security for mobile edge computing," *Future Gener. Comput. Syst.*, vol. 100, pp. 531–541, 2019.

[8] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tut.*, vol. 19, no. 3, pp. 1628–1656, Third Quarter 2017.

[9] E. Cuervo et al., "MAUI: Making smartphones last longer with code offload," in *Proc. Int. Conf. Mobile Syst., Appl. Serv.*, 2010, pp. 49–62.

[10] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. Conf. Comput. Syst.*, 2011, pp. 301–314.

[11] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Proc. Int. Conf. Mobile Syst., Appl. Serv.*, 2012, pp. 59–79.

[12] J. Pan and J. McElhannon, "Future edge cloud and edge computing for Internet of Things applications," *IEEE Internet of Things J.*, vol. 5, no. 1, pp. 439–449, Feb. 2018.

[13] H. Guo, J. Liu, and J. Lv, "Toward intelligent task offloading at the edge," *IEEE Netw.*, vol. 34, no. 2, pp. 128–134, Mar./Apr. 2020.

[14] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, "Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges," *IEEE Commun. Mag.*, vol. 55, no. 4, pp. 54–61, Apr. 2017.

[15] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[16] W. Chen, D. Wang, and K. Li, "Multi-user multi-task computation offloading in green mobile edge cloud computing," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 726–738, Sep./Oct. 2019.

[17] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tut.*, vol. 19, no. 4, pp. 2322–2358, Fourth Quarter 2017.

[18] P. Zhao, H. Tian, K.-C. Chen, S. Fan, and G. Nie, "Context-aware TDD configuration and resource allocation for mobile edge computing," *IEEE Trans. Commun.*, vol. 68, no. 2, pp. 1118–1131, Feb. 2020.

[19] X. Hou et al., "Reliable computation offloading for edge-computing-enabled software-defined IoV," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7097–7111, Aug. 2020.

[20] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci, "ULOOF: A user level online offloading framework for mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 17, no. 11, pp. 2660–2674, Nov. 2018.

[21] M. Golkarifard, J. Yang, Z. Huang, A. Movaghar, and P. Hui, "Dandelion: A unified code offloading system for wearable computing," *IEEE Trans. Mobile Comput.*, vol. 18, no. 3, pp. 546–559, Mar. 2019.

[22] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "DeepWear: Adaptive local offloading for on-wearable deep learning," *IEEE Trans. Mobile Comput.*, vol. 19, no. 2, pp. 314–330, Feb. 2020.

[23] X. Chen, J. Chen, B. Liu, Y. Ma, and H. Zhong, "AndroidOff: Offloading Android application based on cost estimation," *J. Syst. Softw.*, vol. 158, 2019, Art. no. 110418.

[24] X. Chen, S. Chen, M. A. Yun, B. Liu, Y. Zhang, and G. Huang, "An adaptive offloading framework for Android applications in mobile edge computing," *SCIENCE CHINA Inf. Sci.*, vol. 062, no. 008, pp. 110–126, 2019.

[25] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019.
[26] G. A. S. Cassel et al., "Serverless computing for Internet of Things: A systematic literature review," *Future Gener. Comput. Syst.*, vol. 128, pp. 299–316, 2022.
[27] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, 2019.
[28] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with Open-Lambda," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput.*, 2016, pp. 33–39.
[29] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android Java code for on-demand computation offloading," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 233–248, 2012.
[30] Z. Cheng, P. Li, J. Wang, and S. Guo, "Just-in-time code offloading for wearable computing," *IEEE Trans. Emerg. Topics Comput.*, vol. 3, no. 1, pp. 74–83, Mar. 2015.
[31] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter, and G. Min, "Energy-efficient offloading for DNN-based smart IoT systems in cloud-edge environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 683–697, Mar. 2022.
[32] Y. Du, J. Li, L. Shi, T. Liu, F. Shu, and Z. Han, "Two-tier matching game in small cell networks for mobile edge computing," *IEEE Trans. Services Comput.*, vol. 15, no. 1, pp. 254–265, Jan./Feb. 2022.
[33] M. Altamimi, A. Abdrabou, K. Naik, and A. Nayak, "Energy cost models of smartphones for task offloading to the cloud," *IEEE Trans. Emerg. Topics Comput.*, vol. 3, no. 3, pp. 384–398, Sep. 2015.
[34] K. Elgazzar, P. Martin, and H. S. Hassanein, "Cloud-assisted computation offloading to support mobile services," *IEEE Trans. Cloud Comput.*, vol. 4, no. 3, pp. 279–292, Third Quarter 2016.
[35] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, 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./Oct. 2017.
[36] H. Wu, W. J. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1464–1480, Jul. 2019.
[37] P. Civicioglu, "Backtracking search optimization algorithm for numerical optimization problems," *Appl. Math. Comput.*, vol. 219, no. 15, pp. 8121–8144, 2013.
[38] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[39] P. A. Apostolopoulos, E. E. Tsiropoulou, and S. Papavassiliou, "Risk-aware data offloading in multi-server multi-access edge computing environment," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1405–1418, Jun. 2020.
[40] P. K. Bishoyi and S. Misra, "Enabling green mobile-edge computing for 5G-based healthcare applications," *IEEE Trans. Green Commun. Netw.*, vol. 5, no. 3, pp. 1623–1631, Sep. 2021.
[41] Y. Li, D. Guo, Y. Zhao, X. Cao, and H. Chen, "Efficient risk-averse request allocation for multi-access edge computing," *IEEE Commun. Lett.*, vol. 25, no. 2, pp. 533–537, Feb. 2021.

**Xing Chen** (Member, IEEE) received the BS and PhD degrees from Peking University, in 2008 and 2013, respectively. He is a professor with Fuzhou University, and the director of Fujian Key Laboratory of Network Computing and Intelligent Information Processing. He joined Fuzhou University since 2013. He focuses on the software systems and engineering approaches for cloud and mobility. His current projects cover the topics from self-adaptive software, computation offloading, model driven approach and so on. He has published more than 80 journal and conference articles, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Industrial Informatics*, etc. He obtained the Natural Science Fund of Fujian for Distinguished Young Scholars and the Program of Fujian for the Top Young Talents. He was awarded two First Class Prizes for Provincial Scientific and Technological Progress, separately, in 2018 and 2020.

**Ming Li** received the BS degree in computer science and technology from Fuzhou University, Fujian, China, in 2019, where he is currently working toward the PhD degree in computer technology with the College of Mathematics and Computer Science, Fuzhou University. He has also been a part of the Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, since September 2019. His current research interests include system software, and edge computing.

**Hao Zhong** (Member, IEEE) received the MS and PhD degrees from Peking Univeristy in 2005 and 2009, respectively. He is an associate professor with Shanghai Jiao Tong University. His research interest is the area of software engineering. He served on the program committees of reputable venues such as ICSE, ESEC/FSE, ASE, OOPSLA, ICSME, MSR and COMPSAC. He is a recipient of ACM SIGSOFT Distinguished Paper Award, the best paper award of ASE, and the best paper award of APSEC.

**Xiaona Chen** received the BS degree in software engineering from Fuzhou University, Fujian, China, in 2019. She is currently working toward the MS degree in software engineering with the College of Mathematics and Computer Science, Fuzhou University. She has also been a part of the Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, since September 2019. Her current research interests include system software, and computation offloading.

**Yun Ma** (Member, IEEE) received the PhD degree majoring in computer science from the School of EECS, Peking University, under the direction of professor Hong Mei and professor Gang Huang. His research interests lie in mobile computing, Web technologies, and services computing. Currently, he focuses on synergy between the mobile and the Web, trying to improve the mobile user experience by leveraging the best practices from native apps and Web apps.

**Ching-Hsien Hsu** (Senior Member, IEEE) is a chair professor and the dean of the College of Information and Electrical Engineering, Asia University, Taiwan. His research includes high-performance computing, cloud computing, parallel and distributed systems, Big Data analytics, ubiquitous/pervasive computing, and intelligence. He has published 100 papers in top journals such as *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Emerging Topics in Computing*, *IEEE System*, *IEEE Network*, *ACM Transactions on Multimedia Computing, Communications, and Applications* and book chapters in these areas.