Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 00 (2019) 1–13

JSS

ANDROIDOFF:Offloading Android Application Based on Cost Estimation

Xing Chen[a,b], Jiaqing Chen[a,b], Bichun Liu[a,b], Yun Ma[c], Ying Zhang[d], Hao Zhong[e,*]

[a]*College of Mathematics and Computer Science,Fuzhou University, Fuzhou 350116, China;*
[b]*Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou 350116, China;*
[c]*School of Software, Tsinghua University, Beijing 100084, China;*
[d]*National Engineering Research Center of Software Engineering, Peking University, China;*
[e]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China;*

Abstract

Computation offloading is a promising way of improving the performance and reducing the battery power consumption, since it moves some time-consuming computation activities to nearby servers. Although various approaches have been proposed to support computation offloading, we argue that there is still sufficient space for improvements, since existing approaches cannot accurately estimate the execution costs. As a result, we find that their offloading plans are less optimized. To handle the problem, in this paper, given an Android application, we propose a novel approach, called ANDROIDOFF, that supports offloading at the granularity of objects. Supporting such capability is challenging due to the two reasons: (1) through dynamic execution, it is feasible to collect the execution costs of only partial methods, and (2) it is difficult to accurately estimate the execution costs of the remaining methods. To overcome the challenges, given an Android application, ANDROIDOFF first combines static and dynamic analysis to predict the execution costs of all its methods. After all the costs are estimated, ANDROIDOFF synthesizes an offloading plan, in which determines the offloading details. We evaluate ANDROIDOFF on a real-world application, with two mobile devices. Our results show that, compared with other approaches, ANDROIDOFF saves the response time by 8%-49% and reduces the energy consumption by 12%-49% on average for computation-intensive applications.

Keywords: Computation Offloading, Code Analysis, Mobile Edge Computing

1. Introduction

Given the rise of artificial intelligence and big data, mobile applications are becoming both computation and data intensive. Meanwhile, with the rapid development of computing and communication technologies, the computation platform of mobile applications has expanded from smartphones and tablet computers to other devices (*e.g.*, wearable devices [19], vehicles[77], unmanned aerial vehicles [47]). As a result, although the computation capability of mobile devices increases rapidly in recent years, users are complaining the slow response of their mobile applications, especially for those compute-intensive applications [53, 83]. Many factors contribute to the slow response of mobile applications. For example, many obsolete devices are still in use, and an application may be not well configured for such devices [1, 2, 58, 78]. As another example, most devices are powered by batteries, but the power capacity of batteries is still a bottleneck for compute-intensive applications [13, 33, 45].

Computation offloading, *i.e.*, moving intensive computing of an application to nearby servers, is a promising way to improve the performance and to reduce the battery power consumption of a mobile application as well [14, 15, 34,

*Corresponding author

37, 44, 59]. In particular, Mobile Cloud Computing (MCC) has been introduced to extend computing capability and battery capacity of mobile devices, by offloading computation-intensive applications to the cloud. Nevertheless, the network communication between mobile devices and the cloud can lead to significant execution delay. To cope with the delay problem, Mobile Edge Computing [62, 68, 42] (MEC) frameworks have been introduced, which provide servers to nearby mobile devices and enable moving highly demanding computing to such servers. Compared with MCC, MEC has the potential of offering significantly lower latencies.

To fully release the potential of offloading, a mobile application needs to determine which of its parts shall be moved to MEC servers. To achieve this, an application has to determine the execution costs of its parts, and after that, the problem of offloading can be reduced to the traditional optimization problem [54]. In literature, researchers [27, 43, 25] dynamically execute parts of an application to collect the execution costs. Some recent approaches (*e.g.*, [50]) introduce static analysis to determine parts that shall be moved together, but these approaches still need to execute an application to collect the execution costs of its parts. Due to various technical limitations, it is rather difficult to achieve high test coverage, especially for a mobile application [24]. As many parts are not executed, it is infeasible to collect their execution costs, and thus it becomes infeasible to make correct decisions, when offloading such parts.

To handle the problem, in this paper, we propose a novel approach, called AndroidOff, that supports partial offloading of Android applications. Compared with the prior approaches [27, 43, 25, 50], AndroidOff is able **to predict execution costs of methods, even if they are not executed**. Our insight is that **we can collect the execution costs of a subset of all the methods, and then predict the execution costs of the remaining methods, based on static analysis**. To fulfil our insight, we have to overcome two major challenges. First, we have to build a proper model through static analysis, so that we can accurate predict execution costs. Second, we have to design an optimization algorithm, so that we can make correct decisions when offloading applications.

To address the challenges, in this paper, we present a novel offloading approach called AndroidOff. Our paper makes the following major contributions:

- A novel approach, called AndroidOff, that supports offloading Android applications. To predict their execution costs, AndroidOff builds a comprehensive model for classes and methods through static analysis. Given the collected execution costs of a subset of methods, our model can predicts the execution costs of the remaining methods. Furthermore, based on the collected and estimated costs of all parts, AndroidOff uses an optimization function to determine which parts shall be moved to MEC serves.
- An evaluation on a real-world license plate recognition application. Our results show that AndroidOff saves the response time by 8%-49% and reduces the energy consumption by 12%-49% on average for computation-intensive applications. In addition, our results show that the predicting model's fitting degree is 0.786, which is higher than other regression models.

The rest of this paper is organized as follows. Section 2 introduces our offloading framework. Section 3 introduces a motivating example. Section 4 introduces our approach. Section 5 presents our evaluation on a real-world application. Section 6 introduces related work. Section 7 concludes this paper.

2. Preliminary

In our previous work [84], we implement an adaptive framework, called Dpartner, which supports mobile applications with the offloading capability in MCC, and then extend it for MEC [22]. The core of this offloading mechanism is composed of two elements: proxies and endpoints. As shown in Figure 1, `NProxy`, which is a proxy of the callee `N` object. `NProxy` does not do any computation itself, but forwards its invocations to the caller `X` object. The `Endpoint` is responsible for determining the current location of `N` and for the crossing network communication from `X` to `N`. As a result, if the location of `N` is changed, `NProxy` keeps unchanged so that object `X`, will not get noticed. When `N` is running in a remote virtual machine, the `Endpoint` will take advantage of a given Remote Communication Service to get a reference to `N`, and pass it back to `X`. After that, and `X` can use the reference to invoke `N` remotely. When `X` and `N` both run in the same virtual machine, the `Endpoint` will directly obtain the in-virtual-machine reference of `N`, so that `X` can invoke `N` without going through the network stack.

Following the above mechanism, Dpartner has three major steps to offload an application: First, it synthesizes a proxy class and an endpoint for each class in an application. Second, it compiles all the classes and their endpoints

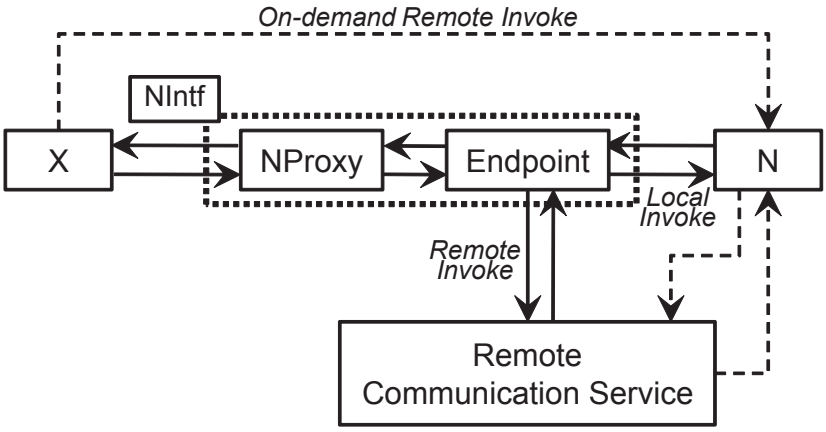


Figure 1: On-Demand remote invocation (target structure)

into a `jar` file, and the compiled file is deployed to a server. Third, it compiles all the classes and the proxies into an Android `apk` file, and the file is deployed to a phone. Please note that source files are not offloaded.

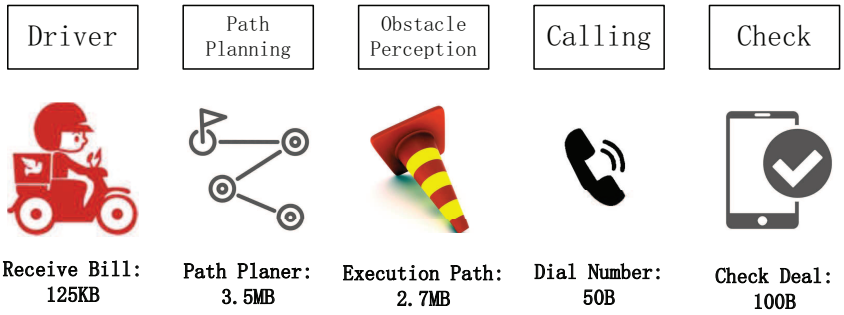
We have provided the mechanisms to offload an application in MCC and MEC. With the support of Departner, AndroidOff offloads at the granularity of objects of Android applications. In our framework, the fields and methods of an object are offloaded together according to the offloading scheme. Thus, an offloaded method can access fields of the same class locally. However, it requires programmers to manually determine which parts of an application shall be offloaded [84] or collect the execution costs of all its parts [22], in order to achieve the optimal offloading decision. AndroidOff proposes an automatic way to determine which parts shall be offloaded. The details of AndroidOff are introduced in Section 4.

3. A Motivating Example

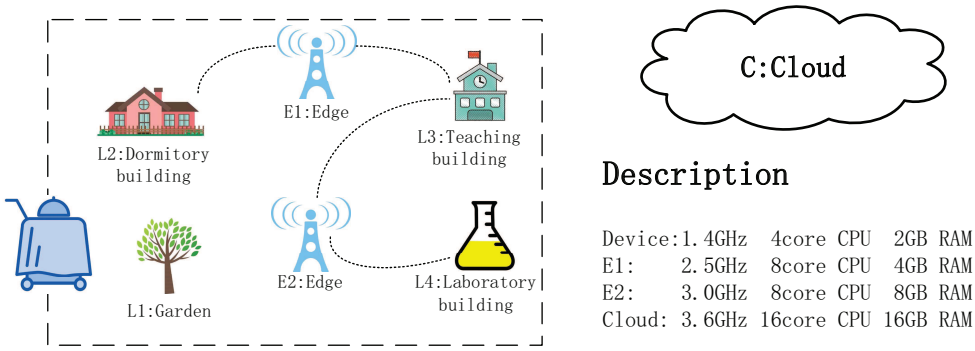
In this section, we use a scenario as shown in Figure 2 to illustrate the challenges and the benefits of AndroidOff. In this scenario, several Unmanned Ground Vehicles (UGVs) are employed to deliver meals in our university. The software of the UGVs is implemented as an Android application. As shown in Figure 2a, the application has five major components such as a driver module, a path planning module, an obstacle perception module, a calling module, and a check module. After a UGV receives a bill, it carries the meal to its customer. As the first step, it has a path planer to decide the path for the delivery. After the path is determined, the driver module and the obstacle perception module execute the path. Upon its arrival, it dials the customer to pick up the meal. After that, it checks whether the bill is paid and the meal is picked up. The computation capability of a UGV is limited. To support the real scale computation, as shown in Figure 2b, we provide three servers as the computation context. The servers are `C`, `E1`, and `E2`, which can be used for offloading in different locations.

Besides Dpartner [84], there are several other frameworks that support offloading [27, 43, 25, 50]. For example, MAUI [27] and ThinkAir [43] deploy servers on cloud and modify all the methods of an application to allow its offloading. In particular, if programmers determine that a method can be offloaded, they have to manually add a `[Remoteable]` annotation or a `@Remote` attribute to the method. At the server side, they provide a linear program solver to determine the optimal partition strategy. To optimize the process, it needs many facts from an application (*e.g.*, the execution time of methods). To the best of our knowledge, all the existing approaches rely on dynamic executions to collect such facts or simply ignore such facts. Due to various limitations, in practice, it is quite difficult to collect such facts through dynamic analysis. For example, it is difficult to construct test inputs to cover specific methods, so their execution time cannot be collected. Without accurate facts, it becomes infeasible to correctly decide which parts of an application shall be offloaded.

The major advantage of AndroidOff lies in that it combines dynamic analysis and static analysis to estimate the execution costs of all the methods. First, we manually prepare test inputs, and execute an application with our



(a) Process of a food delivery application.



(b) Context of the UGV in the campus.

Figure 2: The sample scenario.

test cases to collect the execution costs of some methods. Second, AndroidOff builds weighted call graphs for all the methods. Finally, based on the mined relations between known execution costs and their weighted call graphs, AndroidOff predict the execution costs of all the methods. Besides this advantage, AndroidOff has other benefits such as smaller granularity of offloading and dynamically offloading. For this example, AndroidOff determines that the check module shall be offloaded to remote servers, when UGVs are in our laboratory building where servers are available, but while it shall not be offloaded when UGVs are in our garden. AndroidOff is able to make the correct decision, since all the execution costs are predicted.

4. Approach

Figure 3 shows the overview of AndroidOff. For the nodes, we use rectangles to denote its components and circles to denote its internal data. For the edges, red ones denote data flows, and blue ones denote requests. AndroidOff supports offloading at the granularity of objects. To make offloading decisions, AndroidOff has two major steps. In the first step, AndroidOff trains an estimation model (Section 4.1). Given the execution costs of some methods and the built weighted call graphs of all the methods, the trained model of AndroidOff is able to predict the execution costs of all the methods. Based on the predicted cost, in the second step, AndroidOff synthesizes an offloading decision that minus the overall execution cost for a local application (Section 4.2). In this paper, the execution cost of a method refers to its execution time.

In Figure 3, `Einvoke` denotes the execution cost of a method, and `Sinvoke` denotes the execution cost of a method but subtracts the costs of its called methods. For example, if the `m1` method calls the `m2` methods for four times, $Sinvoke(m1) = Einvoke(m1) - 4 \times Einvoke(m2)$. When we make offloading decision, we use `Sinvoke` instead of `Einvoke`, since its called methods can be executed on other computation nodes. In such cases, the execution cost of a method can change significantly. A more concrete example is described as follows: As shown in Figure 4, in the body of `printMax`, `max` is called for twice, the `Sinvoke` values of `printMax` and `max` are calculated as:

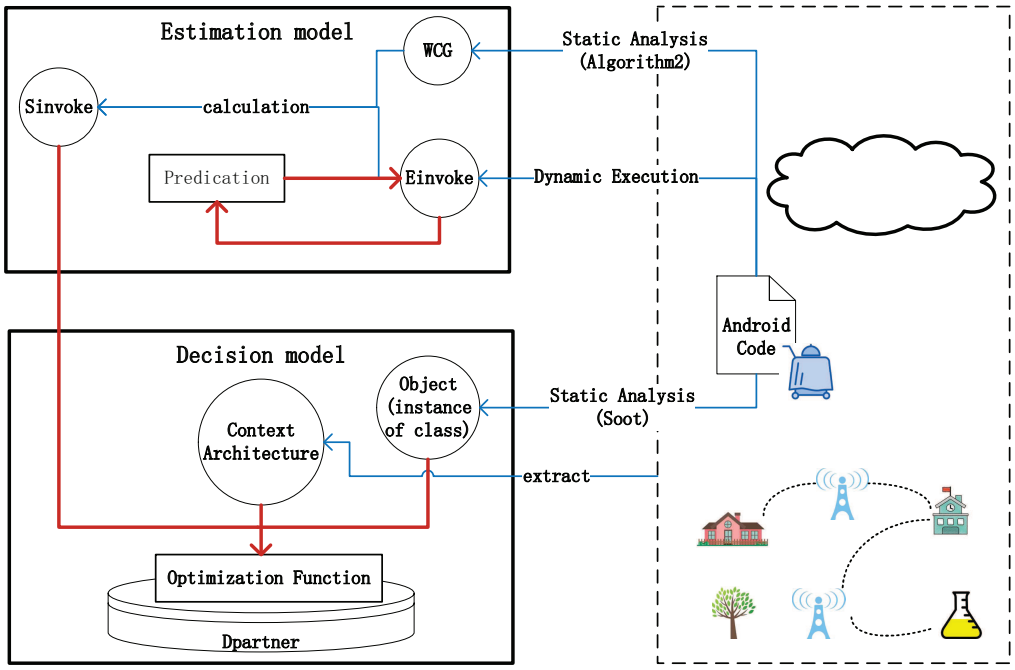


Figure 3: The overview of AndroidOff

$Sinvoke(printMax) = Einvoke(printMax) - 2 \times Einvoke(max)$, $Sinvoke(max) = Einvoke(max)$. With `Sinvoke`, we do not lose information, since we estimate the execution costs for all the methods.

More precisely, Algorithm 1 shows the details of the two above steps. It takes the source code of an application and an environment context as its inputs. In particular, the environment context is modeled as a graph, whose nodes denote computation nodes (*e.g.*, local devices and remote servers) and edges denote communication links between two nodes (*e.g.*, data transmissions and response time). Section 4.2.1 introduces the graph in more details. The output of Algorithm 1 is the optimal deployment, which shows the locations where each object shall be offloaded. First, Line 2 builds weighted call graphs (WCGs) through static analysis. Line 3 executes the application on all the computation nodes to obtain the `Einvoke` of some methods. Line 4 predicts the `Einvoke` of the remaining methods through Random Forest Regression. After that, Lines 7 to 19 compare the response time for all the feasible deployment plans (see Section 4.2.2 for details).

After decision, Dpartner can offload every instance of classes on corresponding compute nodes.

4.1. Estimating Execution Cost

To make the estimation, AndroidOff first builds weighted call graphs (WCGs) for all the methods (Section 4.1.1). After that, it predicts execution costs based on built WCGs (Section 4.1.2).

4.1.1. Extracting Weighted Call Graph for Program

AndroidOff builds an WCG for an Android application, and the definition of an WCG is as follow:

Definition 1. *A weighted call graph is a directed graph $G_P = (M, R)$ representing call relations between methods of a program P, where $M = \{m_1, m_2, \cdots, m_n\}$ is the set of method nodes of P and R is the set of method call edges. Each $r_{ij} \in R$ edge represents a method call from m_i to m_j, and its weight denote the frequencies of the method call.*

Computation offloading decision depends mainly on the response time of application, which is associated with the methods in program. Android applications consist of methods, and there is a `main` method acting as an entry point of the application. From the `main` method, AndroidOff uses Soot [69] to build the WCG. Algorithm 2 shows the

Algorithm 1 The ANDROIDOFF Framework

Input: The source code of an application `code`; A context architecture $G_C = (N, E)$; A set of objects $OBJ = \{obj_1, obj_2, \cdots, obj_n\}$;
Output: An optimal offloading decision $(DEP)_{optimal} = (dep(obj_1), dep(obj_2), \cdots, dep(obj_n))$;

```
1: procedure GENERATE ESTIMATION MODEL
2:     Weighted Call Graph G_P = (M, R) ← Algorithm 2(code)
3:     EinvokeI ← Dynamic execute code on every computation nodes
4:     EinvokeII ← Predication(EinvokeI)
5:     calculate Sinvoke(WCG,Einvoke)
6: end procedure
7: procedure GENERATE DECISION MODEL
8:     (DEP)_optimal ← φ
9:     T_response ← 0
10:    n ← |OBJ|
11:    m ← |N|
12:    for each DEP do
13:        if DEP cannot meet the conditions to communicate or offload in Section 4.2.1 then
14:            do not modify (DEP)_optimal, (T_response)_smallest
15:        else
16:            calculate T_response ← optimizationfunction(Sinvoke, G_C, OBJ)
17:            if (T_response)_smallest = 0 or (T_response)_smallest > T_response then
18:                (DEP)_optimal ← DEP
19:                (T_response)_smallest ← T_response
20:            end if
21:        end if
22:    end for
23:    Dpartner((DEP)_optimal)
24: end procedure
```

process. We use a hash map to store edges. Its keys are in the format of $m_i@m_j$, where m_j denotes the successor of m_i. The set, $U_r = \left\{u_r^1, u_r^2, \cdots, u_r^n\right\}$, denotes all of statements inside m_r, and each $u_r^i \in U_r$ denotes the i*th* statement in U_r. Line 3 iterates U_r. For each element, Line 4 extracts the keywords in u_r^i, with Soot. Here, keywords are the instructions that are defined by Soot. For example, the `JAssignStmt` keyword indicates an `assignment` statement; the `JReturnStmt` keyword indicates the `return` statement of a method; the `JGotoStmt` keyword indicates a `jump` statement; and the `invoke` keyword indicates that a method invocation. The manual of Soot [6, 18] present the complete definitions of these keywords. For each method invocation, Lines 6 to 14 update the `M` and `R` sets according to keywords. In particular, if u_r has a keyword that indicates a call to the m_s method (*e.g.*, `invoke.ms(parameter)`), we determine that the m_r method calls the m_s method. In such a case, we add the m_s method to the M set. In addition, we look up the key of hash map R, if the edge $m_r@m_s$ already existed, the corresponding value r_{rs} add one, else record the edge into hash map R. After updating, enter the callee m_s and do Line 3 again.

Besides the call relations, for each method, ANDROIDOFF extracts features such as block depths, the number of statements, and its complexity:

Definition 2. *$m_i =< blockDepth, percentBranchStatements, complexity, statements, calls >$: blockDepth denotes the depth of function, which is calculated as the nesting levels of branches in m_i. percentBranchStatements denotes the proportion of branch statements, which is branch statements such as `if`, `for`, `while` and `switch` over the total statements. complexity denotes the number of paths in m_i, which is calculated as:*

$$complexity = e - n + 2 \tag{1}$$

where e is the number of edges and n is the number of nodes . statements denote the number of statements and calls denote the times of external invocations in m_i.

```
public void printMax(int a,int b){
       int c = max(a,b);
       System.out.println("The maximum between
"+a+"and "+b+" is"+c);
       int d = max(666,999);
   }
   public int max(int num1,int num2){
       int result;
          if (num1 > num2)
            result = num1;
          else
            result = num2;

          return result;
   }
```

Figure 4: An example of method invocation.

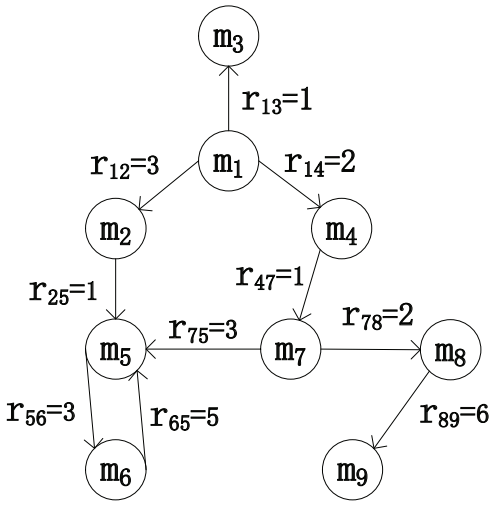


Figure 5: An example of weighted call graphs.

4.1.2. Predicting Execution Costs

If the m_i method is executed on the n_k node, we define its execution cost as follows:

Definition 3. *$Einvoke_{n_k}^{m_i} =< Etime, Edatasize >$: the Etime time denotes the execution time from first statement to last statement, and Edatasize denotes the amount of data transmission.*

To guarantee the validity of our recorded *Einvoke* values, we execute each program ten times and take the average as its *Einvoke* value. Here, we calculate the relative error between the ten values and their average value. If the relative error of a value is greater than 10%, we determine that the value is accidental. When this happens, we recalculate the average, after we remove the accidental values. The execution costs are different on different nodes. To handle this issue, we execute the program on all the nodes. It is worth noting that we could only record 80% of methods after executing the program. The execution costs of the remaining 20% methods are estimated according to our static analysis and the costs of those executed methods.

A simple linear regression model can be insufficient to reveal the relation between features and execution costs. ANDROIDOFF uses the Random Forest Regression Algorithm [39] of WEKA [28], which is effective to handle imbalanced data with satisfactory robustness. Brieman [20] proposed Random Forest (RF) and proved it can carry out the

Algorithm 2 The WCG Generation Algorithm

Input: A main method m_r,its statement $U_r = \left\{u_r^1, u_r^2, \cdots, u_r^n\right\}$
Output: A weighted call graph $G_P = (M, R)$;

$M \leftarrow M + m_r$
$R \leftarrow \phi$
for each $u_r^i \in U_r$ **do**
    $keywords \leftarrow Soot(u_r^i)$
    if $\exists$ *"invoke"* $\in keywords$ **then**
        $m_s \leftarrow getMethodName(keywords)$
        $M \leftarrow M + \{m_s\}$
        if *"*$m_r@m_s$*"* $\in R.key$ **then**
            $++r_{rs}$
        else
            $r_{rs} \leftarrow 1$
            $R \leftarrow R + r_{rs}$
        end if
        $m_r \leftarrow m_s$
    end if
end for

nonlinear relation between the variables. It is a non-linear model-building tool, which is widely used in some popular fields such as data mining and bioinformatics [79, 26]. Moreover, RF can give the ranking for the importance of the variables, and the error rate for out of bag (OOB) data can give a good estimation for generalization ability of RF. Therefore, RF is a good tool to encode the nonlinear relation between the features of methods and execution costs, and will be effective for predicting those remaining methods.

Random Forest Regression(RFR) algorithm:

Random forest for regression consists of a collection of regression trees $\{h(X, \theta_k), k = 1, \cdots, K\}$, where `X` is the observed input vector and θ_k are independent, identically distributed random vectors. `Y` as output prediction values are numerical. The training data is assumed to be independently drawn from the joint distribution of `(X,Y)`, the random forest prediction is unweighted average of all regression tree prediction:$\bar{h}(X) = (1/K)\sum_1^K h(X, \theta_k)$, where `K` is the number of regression models of the decision tree for each sample to get `K` regression prediction results. The process of random forests regression algorithm is as follows [20]:

1. If the number of cases in the training set is n , sample b cases randomly by using bootstrap and grow b regression trees accordingly. Each time the samples that are not selected form b out-of-bag data are used as random forests test samples.

2. If there are p variables in the original data, a number of $mtry$ variables $mtry \leq p$ are specified at each node of a regression tree as alternate branch variables. Then select the best branch according to the branching optimum rule.

3. Each tree then begins to branch recursively from top to bottom. Set the node with the least size as the condition for the tree to stop growing.

4. Build a random forests model with b regression trees generated. Its effect is assessed with residual mean square, which is predicted by out-of-bag (OOB) data, as shown in Formula 2 and 3, where y_i is the actual value of the dependent variable in OOB, $\check{y}_i$ is the predicted value by random forests to OOB and $\check{\sigma}_y^2$ is the variance by random forests to OOB predicted value.

$$MSE_{OOB} = n^{-1}\sum_1^n \left\{y_i - \check{y}_i^{OOB}\right\}^2 \tag{2}$$

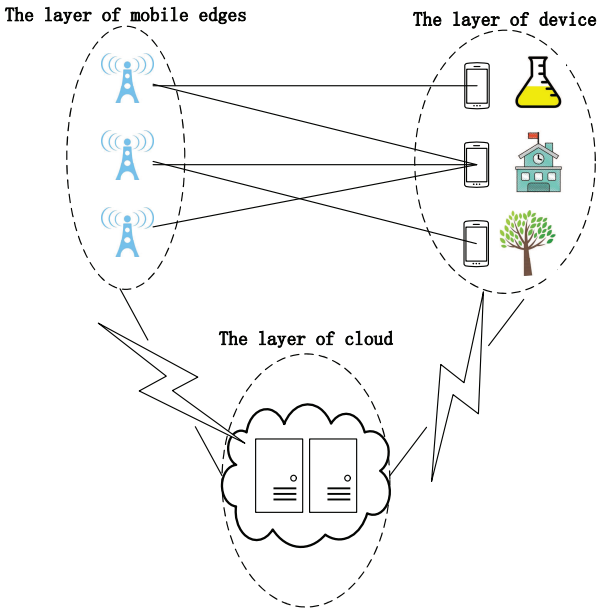


Figure 6: Context architecture.

$$R^2_{RF} = 1 - \frac{MSE_{00B}}{\check{\sigma}^2_y} \tag{3}$$

Definition of prediction model:

A regression model for predicting execution costs is obtained by training RF with the important variables and the parameters `Ntree`, `Mtry`. Thus, the final model is defined as follow:

(1) The variables: blockDepth, percentBranchStatement, complexity, statements, calls, Parm

(2) The type of a random forest: regression

(3) The number of regression trees grown based on training data(`Ntree`): 50

(4) The number of different predictors tested at each node(`Mtry`): 3

(5) The mean of squared residuals(MSE_{OOB}): 200.76 ms

In training procedure, when building the training data, for each collected execution, we extract its features as listed in Definition 2, and consider its execution cost as the label. Set $Parm = \{Parm_1, Parm_2, \cdots, Parm_n\}$ denotes parameter types in m_i. And the value of $Parm_k \in Parm$ is collected by code analysis, which denotes the number of $Parm_k \in Parm$. We take parameter types into consideration, since a complicated type often leads to more data transmissions among computation nodes. Our evaluation results show that this feature is the best one for predicting. After the model is trained, AndroidOff can estimate the execution costs of the remaining methods (*i.e.* $Einvoke^{m_i}_{n_k} =< Etime, Edatasize >$).

Formally, we define the `Sinvoke` cost of a m_i method in the n_k computation node as follows:

Definition 4. *$Sinvoke^{m_i}_{n_k} =< Stime, Sdatasize >$: the time Stime denotes the execution time and Sdatasize denotes the amount of data transmission except external invocations in m_i executed in n_k,which should be calculated by:*

$$Sinvoke^{m_i}_{n_k} = Einvoke^{m_i}_{n_k} - \sum^{m_j \in Post(m_i)} Einvoke^{m_j}_{n_k} * r_{ij} \tag{4}$$

where r_{ij} denotes the frequencies of a method call from m_i to m_j and $Post(m_i)$ denotes all the successors of m_i.

For a method (m_j), all its called methods are extracted through static analysis. As these called methods can be offloaded to other computation nodes, their execution costs shall not be included in the local execution cost of m_j, *i.e.*,

Table 1: The factors that can influence the offloading decision

Symbol	Description
OBJ	the set of core movable objects
$INVOKE$	the set of call relations between objects
N	the set of compute nodes including DS, ME and RC
$n_k \in N$	the compute nodes n_k
$v_{n_i n_j}$	the data transmission rate between n_i and n_j
$rtt_{n_i n_j}$	the round-trip time between n_i and n_j
T_{obj_i}	the total offloading time of obj_i
$T_e(obj_i)$	the execution time of obj_i
$T_d(obj_i)$	the data transmission time of obj_i
$T_{response}$	the response time of application
$Sinvoke_{n_k}^{m_i}$	the execution cost except external invocations of m_i in n_k
$invoke_{pq}$	a object call from obj_p to obj_q
$dep(obj_i)$	the offloading node for obj_i

$Einvoke_{n_k}^{m_i}$. To handle this issue, the execution costs of called methods are subtracted from $Einvoke_{n_k}^{m_i}$, which is equal to the sum of each $m_j \in Post(m_i)$. During rumtime, the actual cost of $m_j \in Post(m_i)$ is calculated as its $Einvoke_{n_k}^{m_j}$ multiplied by the weight of r_{ij}, *i.e.*, the frequencies of the method call. Algorithm 2 describes the details.

As a result, the execution cost except external invocations of methods in all compute nodes(*i.e.*, $Sinvoke_{n_k}^{m_i} =< Stime, Sdatasize >$) could be achieved.

4.2. Offloading with Estimated Costs

In this section, we presents our estimation model for offloading, and the model calculates the response time of each offloading decision. It determines the minimum cost of all the decisions as the optimal offloading. We first introduce the factors that affect the offloading decision and a context model that describes the environment (*e.g.*, computation nodes). After that, we introduce our fitness function.

4.2.1. Contributory Factor

In the offloading-decision process, AndroidOff determines which objects shall be moved, and which compute nodes shall be moved to. The optimized decision shall minimize the overall execution cost of a program.

Figure 6 shows our context architecture, which consists of a device in different scenarios (DS), several mobile edges (ME) and a remote cloud (RC). We use a graph to present this network $G_C = (N, E)$, where N denotes a set of compute nodes including local device and remote servers, and E represents a set of communication links among nodes $n_i \in N$. Each $edge(n_i, n_j) \in E$ is associated with a data transmission rate $v_{n_i n_j}$ and a round-trip time $rtt_{n_i n_j}$ between n_i and n_j. A typical offloading scenario is as follow: First, objects are created in a local device (the only n_{DS}). In this location, the created objects can be moved to computation nodes (some nodes of n_{ME}) and the remote cloud (n_{RC}). When device moves to the other location, objects can be moved among the local device, the nearby computation nodes and the remote cloud according to decision. As mobile edges are not connected in our setting, if transmission occurs between two objects, they will not be deployed in different n_{ME}.

Table 1 shows our factors for predicting which objects shall be offloaded. Among them, $n_k \in N, v_{n_i n_j}$ and $rtt_{n_i n_j}$ are defined before. We next introduce $DEP = (dep(obj_1), dep(obj_2), \cdots, dep(obj_n))$, where DEP denotes the offloading decision. Each $obj_i \in OBJ$ has an offloading node $dep(obj_i) \in N$. Let $T_e(obj_i)$ denotes the total execution time of obj_i and let $T_d(obj_i)$ denotes the data transmission time of obj_i in $dep(obj_i)$. The response time of application can be represented by $T_{response}$, which is equal to the sum of $T_(obj_i)$. *i.e.*, the total offloading time consisting of $T_e(obj_i)$ and $T_d(obj_i)$. In addition, a fitness function is constructed to calculate $T_{response}$ and evaluate the offloading decision.

Table 2: The device contexts in different locations

	Garden	Playground	Teaching Building	Laboratory Building
E1	X	RTT = 40ms V = 1.5Mb/s	RTT = 40ms V = 1.5Mb/s	X
E2	X	X	RTT = 70ms V = 1Mb/s	RTT = 40ms V = 1.5Mb/s
Cloud	RTT=200ms V=200Kb/s	RTT=200ms V=200Kb/s	RTT=200ms V=200Kb/s	RTT=70ms V=1Mb/s

4.2.2. Optimization Function

Based on the factors in Section 4.2.1, we construct the optimization function as shown in Formula 5. Here, we consider that a decision is optimal, if its fitness value is the smallest.

$$T_{response} = \sum_{obj_1}^{obj_n} T(obj_i), \forall obj_i \in OBJ \tag{5}$$

where $T(obj_i)$ is determined by:

$$T(obj_i) = T_e(obj_i) + T_d(obj_i) \tag{6}$$

Then we have the following calculations for $T_e(obj_i)$ and $T_d(obj_i)$:

$$T_e(obj_i) = \sum \left(S invoke_{dep(obj_i)}^{invoke_{pi}.callee}.S time * invoke_{pi}.invokeTimes \right) \tag{7}$$

$$T_d(obj_i) = \sum \left(\left(\frac{S invoke_{dep(obj_i)}^{invoke_{pi}.callee}.S datasize}{v_{dep(obj_p)dep(obj_i)}} + rtt_{dep(obj_p)dep(obj_i)} \right) * invoke_{pi}.invokeTimes \right) \tag{8}$$

where $obj_p \in OBJ$ is the caller of obj_i and $invoke_{pq}$ should satisfy the following constraint:

$$q = i, \forall invoke_{pq} \in INVOKE \tag{9}$$

The formulation description is expounded as follows: We traverse the `INVOKE` set and find the relationship $invoke_{pq}$, where obj_q is obj_i. For each $invoke_{pi}$, we obtain `Stime` of the callee method on the compute node $dep(obj_i)$ in Section 4.1. After that, $T_e(obj_i)$ is calculated as the sum of `Stime` multiplied by `invokeTimes` in each $invoke_{pi}$ as shown in Formula 7. A single data transmission time between obj_p and obj_i consists of two parts. One part is the `Sdatasize` of the callee in the compute node $dep(obj_i)$, which is obtained in Section 4.1, divided by $v_{dep(obj_p)dep(obj_i)}$. The other is the round trip time between two compute nodes, which are the locations of obj_p and obj_i. Thus, $T_d(obj_i)$ is the sum of single one multiplied by `invokeTimes` in each $invoke_{pi}$, as shown in Formula 8.

As a result, $T(obj_i)$ could be calculated and the accumulation of each $T(obj_i)$ is the response time of application. The result can help to make offloading decision in Algorithm 1.

5. Evaluation

We implemented AndroidOff, and conducted evaluations to explore the following research questions:

(RQ1) To what degree does AndroidOff improve the state of the art (Section 5.1)?

(RQ2) Which parameters are the best? (Section 5.2)?

(RQ3) Which features matter in the prediction? (Section 5.3)?

(RQ4) To what degree does AndroidOff makes correct predictions (Section 5.4)?

(RQ5) How effective is our internal Random Forest compared with other classifiers (Section 5.5)?

For RQ1, our results show that AndroidOff saved 8%-49% response time and reduced 12%-49% energy consumption on average. For RQ2, we find that the best results are achieved when ntree is 50 and mtry is 3. For RQ3, we present the rank of features, as far as their importance is considered. For RQ4, our results show that the model's fitting degree is 0.786. For RQ5, our results show that the fitting degree of RF is higher than the other regression models.

5.1. RQ1. The Improvement Over The State Of The Art

5.1.1. The Setting

In our evaluation, the network context consists of five computation nodes such as two mobile devices and three remote servers, which is an MEC environment. MEC is different from MCC, since MEC includes edge nodes, which provide computing power for mobile terminals. In particular, MCC has only a strong cloud center, but the connection between the center and mobile devices is relatively poor. For MEC, edge nodes provide weaker computing power, but have better network connection to nearby mobile devices. We have four locations such as the garden, the playground, the teaching building and the laboratory building. Table 2 lists the connections among our computation nodes. The column and the row of a cell denotes the round-trip time and the data transmission rate between a mobile and corresponding nodes. We obtain the two measures with the WLAN RTT tool [7]. A smaller `rtt` and a higher `v` denotes a better signal strength.

The two mobile devices include two mobile phones: 1) Honor MYA-AL10 [3] with 1.4GHz 4 core CPU, 2GB RAM, and 2) Honor STF-AL00 [4] with 2.4GHz 4 core CPU, 4GB RAM.

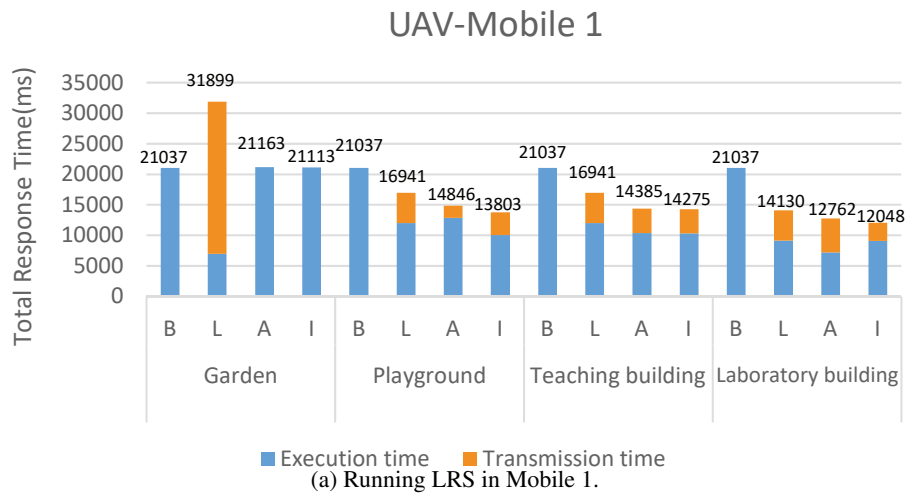
The three remote servers include two mobile edges (`E1` and `E2`) and one cloud, which can be used for offloading in different locations. `E1` is a server with 2.5GHz 8 core CPU and 4GB RAM, and it provides the wifi connections to the playground and the teaching building. `E2` is a server with 3.0GHz 8 core CPU, 8GB RAM, and it provides the wifi connection to the teaching building and laboratory building. The cloud is a server with 3.6GHz 16 core CPU and 16GB RAM, which is accessed from all the locations.

We use an Android application in the evaluation. It is a License Plate Recognition System (LRS). We installed the application on our mobile devices. With both devices, we walked around the garden, the playground, the teaching building, and the laboratory building; record the video continuously of parked cars; and identified their license plates. There are five main computation tasks in this application: shooting, framing, preprocessing, ocr processing, and information storing. The LRS translated a video into images, frame by frame. From these images, the LRS recognizes the plate number and stores it locally. The application needs to connect with other servers to check whether a plate number is legal or not. We went through the whole process for ten times. To reduce the total execution time and the transmission time, we transformed images in grey before being transmitted to the server according to Wang *et al.* [70].

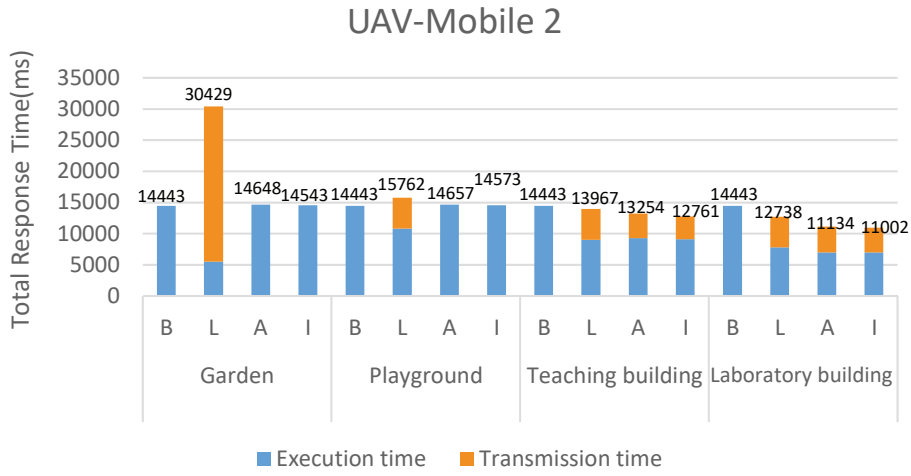
Because we concern whether the difference between AndroidOff and other approaches is significant, Mann-Whitney U test is used to compare whether AndroidOff saved response time and reduced energy consumption significantly. As introduced by McKnight and Najab [52], the Mann-Whitney U test is more general than many other tests, since it does not require normal distribution.

5.1.2. Compared Approaches

In our evaluation, we compared AndroidOff with the three other approaches such as the baseline, the ideal approach and Logger [17]. In particular, for the baseline, the application is executed on mobile device locally, without any offloading and scheduling. For the ideal approach, all the movable objects in application are offloaded with fastest scheme after executing all object deployment in reality. The ideal approach is infeasible in practical, since it needs to exhaustively iterate all the paths and the effort is often unacceptable. Indeed, the exhaustive search before the evaluation takes non-trivial effort. For example, for the ideal approach, when we move a phone inside the teaching building, we have to execute 243 times to compare the response time among 243 possible deployments. Each deployment takes more than 20,000 ms, and those worse cases take even more time. We introduce the ideal approach to illustrate how close AndroidOff is to the ideal one. The goal of Logger is to reduce the computation costs with the lowest overhead of data transmission. As a result, Logger offloads all the movable objects to nearby servers, according to their network connections (*i.e.*, the smallest `rtt` and the highest `v`).



(a) Running LRS in Mobile 1.



(b) Running LRS in Mobile 2.

Figure 7: The performance comparison in four locations

5.1.3. *Measurement*

To show the effectiveness of ANDROIDOFF , we define the following metrics:

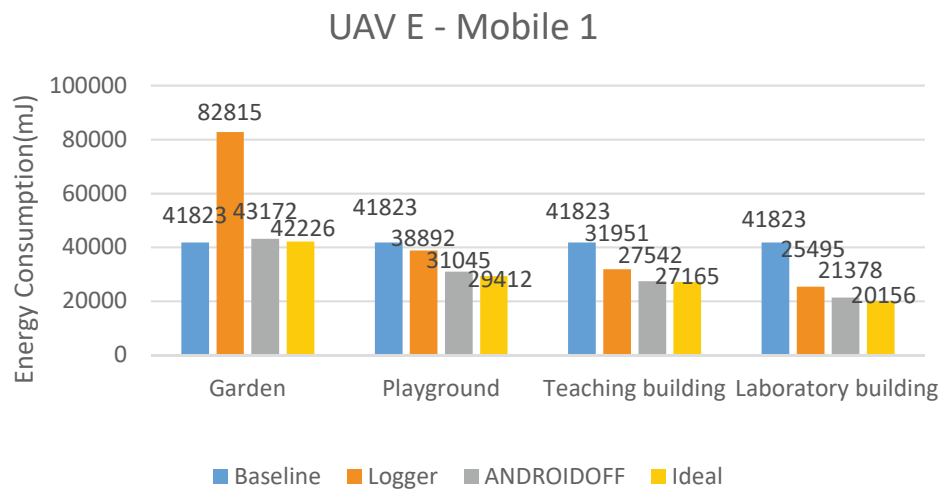
Total response time: We use the total response time of executing the five tasks in application as the metric for performance. To make a fair comparison, we executed the five tasks for ten times, and calculate their averages for comparison. Here, the start time is recorded when the shoot button is pressed and the end time is recorded when parking plate numbers are recognized. It includes execution time and transmission time. The less response time indicate better results.

Execution time: This is the time to compute tasks process on the mobile device or remote servers. The computation on remote servers is usually more efficient than mobile device.

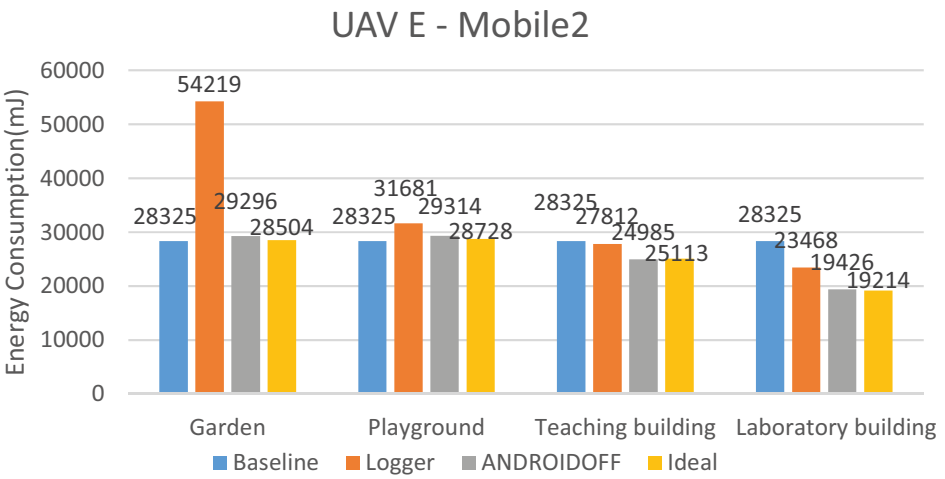
Transmission time: This is the time to transmit data among computation nodes, and it is often slow under poor network connection [71]. The transmission time should be minimized.

Energy consumption: We measure the energy consumption with the PowerTutor [5]. The tool records the details of the energy consumption for each target application. Intuitively, the lower energy consumption indicate better results.

P value: We reject H_0 and accept H_1 only when p is less than 0.05. On the contrary, H_0 is accepted and H_1 is



(a) Running LRS in Mobile 1.



(b) Running LRS in Mobile 2.

Figure 8: The energy consumption comparison in four locations.

rejected when $p \geq 0.05$. Here p is a value without units, which is used to compare with the critical value "0.05". The hypothesis is usually defined as follows:

-(H_0) The difference between two approaches is not statistically significant.

(H_1) The difference between two approaches is statistically significant.

5.1.4. Results

In this section, the compared approaches will be measured according to the metrics from Section 5.1.3.

1. Total response time. The total response time consists of the execution time and the transmission time. Figure 7 shows the time of the compared approaches in the four locations. As the response time includes the time of shooting, framing, preprocessing, ocr processing, and storing, it is reasonable to take 15s-35s. "B", "L", "A", and "I" denote the baseline, the Logger, the ANDROIDOFF and the ideal respectively. For each approaches, the blue bar denotes the execution time and orange one denotes the transmission time. Compared with the baseline on Mobile 1, ANDROIDOFF reduces 29%, 32% and 39% in the playground, the teaching building, and the laboratory building respectively. In addition, compared with baseline, ANDROIDOFF reduces 8% and 23% respectively on Mobile 2. The performance of

Table 3: The differences among offloading approaches

	the ideal approach	Logger	the baseline
AndroidOff	0.6454	0.0274	0.0060
the ideal approach		0.0218	0.0075
Logger			0.7553

Logger is better than the baseline, except in the garden. In this location, movable objects are all offloaded to the Cloud, and the overhead of transmission between the device and the Cloud is expensive. In particular, according to Table 2, Logger offloads objects to Cloud in the garden, but offloads objects to E1 or E2 in other locations. In contrast, based on the trade-off between execution time and network delay, AndroidOff decides to execute such objects locally. Compared with Logger, AndroidOff reduces the response time by 34% on Mobile 1, and 52% on Mobile 2 in the garden. Specially, the total time of AndroidOff is slightly more than the baseline, when offloading is not needed. Because AndroidOff runs locally as the baseline does, its performance is close to the baseline. However, it takes 200 more ms, which are consumed by the proxies. For both devices, the total response time of AndroidOff is close to the ideal approach, with a gap of only 100 ms.

2. Execution time. In Figure 7, the blue bars denote the execution time. They are affected by the performance of computation nodes ($Cloud > E1 > E2 > Mobile1 > Mobile2$). The baseline takes more execution time than others in the playground, the teaching building, and the laboratory building, because it does not offload any objects. In the garden, as Logger offloads more objects than other approaches, it takes the shortest execution time. As the execution time in 7a is longer than 7b, the execution time is significantly shorten on the low-end device.

3. Transmission time. The transmission time depends on the data sizes and the network connection between computation nodes. For Logger, the same amount of data (offloaded objects) are transmitted slower with the decreasing of the network connection. Although the execution time of Logger is the shortest, its transmission time is five times longer than our execution time.

4. Energy consumption. The energy consumption of AndroidOff is less than the baseline and Logger while offloading, as shown in Figure 8. The enerygy consumption decreases by 26%-49% on Mobile 1 and by 12%-31% on Mobile 2. However, Logger reduce only 39% and 17% respectively. When we use Mobile 2 in the garden and the playground, the energy consumption of the ideal is close to that of the baseline, albeit with an overhead of approximately 300 mJ, which originates from the proxies. While Logger will consume more energy in transmitting when the network connection is poor.

5. AndroidOff is significantly different from the baseline and Logger. The first row of Table 3 shows the results. In Table 3, the column and the row of a cell denote two compared offloading approaches. For each cell, the grey background indicates that $p < 0.05$ and H_1 is accepted. The results show that AndroidOff significantly improves the results of the baseline and Logger, since their p values are less than 0.05. The results confirm the effectiveness of AndroidOff, and highlight the importance of the offloading decision.

6. H0 is accepted and AndroidOff is close to the ideal approach. In Table 3, for the first cell, $p > 0.05$ and H_0 is accepted. The result shows that the difference between AndroidOff and the ideal approach is not statistically significant. However, we reject H_0 for the second row. Therefore, the baseline and Logger still have a gap compared with the ideal approach while the effectiveness of AndroidOff is close to it.

In summary, our evaluation results show that AndroidOff is able to extend the battery lifetime of mobile device and reduce the overall response time. Our appendix lists more details of our offloading plan.

5.1.5. Generalization

To show the generalization of our results, we add another application called Voice Recognition System(VRS) as our subject. The application recognizes textual contents from voices. The network context, locations, devices and remote servers are the same as described in Section 5.1.1. There are three main computation tasks in this application: a preprocessor, a recognizer, and a decoder. The data size of VRS is smaller than LRS, since they contain only voices. We compare our approach with the baseline and Logger. In particular, we change the input data size, and observe the changes of the total response time. Figure 9 shows the results in the laboratory building on Mobile 1. The red lines represent the results of LRS, and the blue lines represent the results of VRS. The solid lines represent the results of

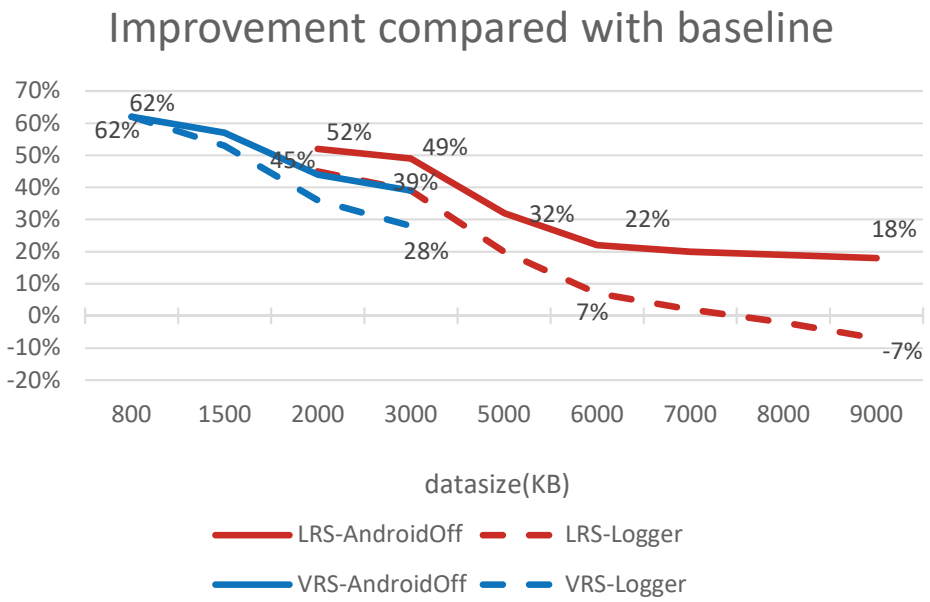


Figure 9: The improvement of total response time compared with the baseline in laboratory building with Mobile 1 and two applications respectively.

ANDROIDOFF , and the dotted lines represent the results of Logger. The figure shows that there is a notable decrease with the increasing of the input data size. ANDROIDOFF reduces 39%-62% for the results of VRS, and the improvement for LRS is even more significant, due to the smaller amount of transmission data. Hence, ANDROIDOFF achieves better results, when the data size of applications becomes smaller. Logger reduces 62% response time when the input data size is small, but it reduce only 28% when the input data size becomes larger (3000KB) in VRS. Furthermore, there is a sharp decline from 3000KB to 9000KB in LRS, and the results of Logger even become negative. LRS requires a large data size, and the results of ANDROIDOFF decrease to about 20% in 6000 KB, since a larger input data size leads to longer transmission time. However, between 6000 KB and 9000 KB, its decrease is at a much slower pace than Logger. Because the baseline has to take longer execution time with large data size while ANDROIDOFF can make a trade-off and transmission time will not become bottleneck. The improvement is relatively stable at 18%. ANDROIDOFF is able to deal with applications with a large input data size, where Logger is less effective.

5.2. RQ2. The Best Parameters

5.2.1. The Setting

To prepare the data of this evaluation, we execute License Plate Recognition System (LRS) ten times and calculate their averages to record Einvoke and extracts features by ANDROIDOFF in Section 4.1.1. Table 4 shows the results of some sample methods. Column "ClassName" lists class names. "ClassType" lists whether the class can be offloaded. Column "M" the ids of methods. Column "MethodName" lists the names of methods. Column "Complexity" lists the number of paths in methods. Column "Statements" lists the number of total statements in methods. Column "blockDepth" lists the depth of methods. Column "Calls" lists the times of external invocations in methods.

The inputs (X) include the blockDepth (B), percentBranchStatement (PE), complexity (CO), statements (S), calls (CA), $Parm = \{int, double, String, int[], double[]\}$ (PA) of method. The outputs (Etime and Edatasize) are denoted as the predicted values (P) of execution costs. In total, we collected the execution costs of 94 methods. For example, Table 5 shows one of the items. In method "onCreate", the complexity is 3, the number of statements is 14, the depth is 5 and the calls is 12, which are shown in the first row of Table 4. The percentBranchStatement is 0.3258 since the number of branch statements is 5. $Parm = \{1, 0, 2, 0, 0\}$ denotes that the number of parameter type "int" is 1 and "String" is 2. The $P = \{20ms, 0.89Kb\}$ denotes the Einvoke we have recorded in Section 4.1.2. We randomly split

Table 4: The features of methods

ClassType	ClassName	M	MethodName	Complexity	Statements	MaximumDepth	Calls
Anchored	UAV	m_1	onCreate	3*	14	5	12
Anchored	Patrol	m_2	capture	4*	8	5	5
		m_3	getMap	3*	9	4	6
		m_4	patrol	2*	2	3	1
Movable	ImageProcess	m_5	imageProcess	2*	4	3	3
Anchord	TakePhoto	m_6	takePhoto	3*	8	4	5
Movable	GrayProcess	m_7	grayProcess	9*	15	6	6
		m_8	resize	5*	9	6	5
Movable	OcrProcess	m_9	getTarget	5*	10	6	3
		m_{10}	ocrAnalysis	6*	13	4	6
Movable	Store	m_{11}	store	1*	2	2	2
Anchored	Fly	m_{12}	down	2*	4	2	2
		m_{13}	up	2*	4	2	2
		m_{14}	left	2*	4	2	2
		m_{15}	right	2*	4	2	3
		m_{16}	speedup	2*	4	2	2
		m_{17}	speedDown	2*	4	2	2
Movable	PathPlanning	m_{18}	A^*	10*	18	7	10

Table 5: A sample item

Sample No	B	PE	CO	S	CA	PA	P
1	5	0.3258	3	14	12	{1,0,2,0,0}	{20ms,0.89Kb}

the 94 data items into two categories: 70% for training and 30% for testing. The training set was used to train the prediction model, and the testing set was used to test the quality of our model.

Two parameters need to be optimized in RF: `ntree`, the number of regression trees grown based on a bootstrap sample of the collected methods; `mtry`, the number of different predictors tested at each node. To find ntree and mtry values that can best predict the uncollected methods, the two parameters (mtry and ntree) were optimized based on the mean squared error (MSE) of calibration. `ntree` values from 10 to 210 with intervals of length 40 were tested, and `mtry` was tested from 1 to 5. The parameter values (`ntree` and `mtry`) were optimized to find the values that could best predict the `Etime` and `Edatasize`. The `ntree` and `mtry` values that yielded the lowest MSE were selected. The RF regression model was conducted as described in [55].

5.2.2. The Measurement

We regard MSE as the evaluation measure of Random Forest Regression Algorithm. Mean Squared Error (MSE) is an average of the squares of the difference between the actual observations and those predicted:

$$MSE = \frac{1}{N}\sum_{t=1}^{N}(observed_t - predicted_t)^2 \tag{10}$$

5.2.3. The results

The results of random forest parameters (`ntree` and `mtry`) are shown in Figure 10, which clearly indicates that RF random forest parameters (`ntree` and `mtry`) affect the error of prediction. The optimization was done using the calibration dataset (n=65) and MSE. The result $ntree = 50$ and $mtry = 3$ yielded the lowest MSE (200.76 ms).

According to Brieman [20], the default value of *ntree* is 500, and the default value of *mtry* is 2. However, Figure 10 shows $mtry = 2$ is not the optimal point. When we use the default values $mtry = 2$, MSE is about twice longer than

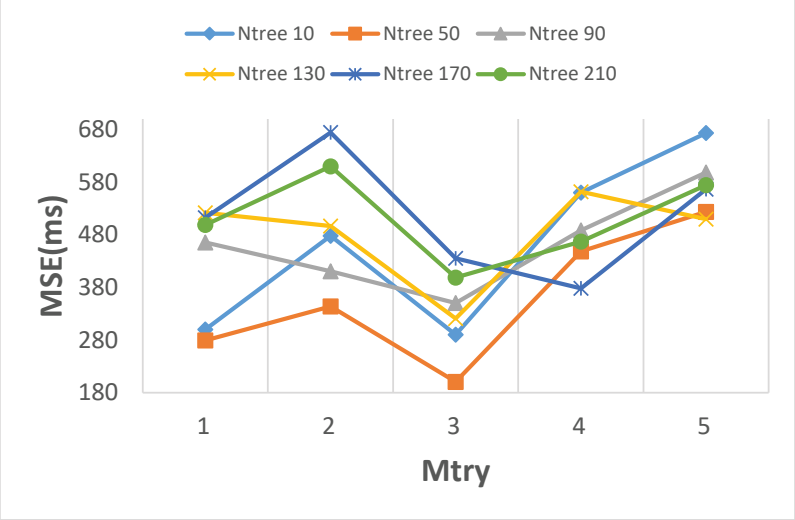


Figure 10: Optimization of random forest parameters using MSE. The optimal ntree and mtry yielded the lowest MSE.

our result *mtry* = 3. Besides, we found the results are convergence and nearly stable when *ntree* = 50, but it takes long time to predict when *ntree* = 500. In this case, we choose *ntree* = 50 and *mtry* = 3 as the best parameters.

5.3. RQ3. The Importance of Features

5.3.1. The Setting

In this evaluation, we reuse the dataset as introduced Section 5.2.1. As in the previous section, we find that the best results are achieved when `mtry` is 3 and `ntree` is 50, we set the two parameters as 3 and 50, respectively.

5.3.2. Measurement

The Out-of-bag (OOB) estimation is a method of measuring the prediction error of random forests and is the mean prediction error on each training sample [40]. The OOB estimates of error rate were used to measure the importance of features. The random forest model was able to explore and rank the features by their importance in estimating Edatasize. Since the feature importance measured in terms of the increase of OOB error, which represents the deterioration of the predictive performance of the model when each feature is permuted.

5.3.3. Results

Figure 11 shows the feature importance measured in terms of the increase of OOB error. The %OOB error of `Parms` as 20.8% is higher than other features. Therefore, `Parms` as the predictors contributed most to the estimation of `Edatasize`. And we take `Parms` into consideration when build the predication model for `Edatasize`.

5.4. RQ4. The Effectiveness of Predicting Execution Costs

5.4.1. The Setting

In this evaluation, we reuse the dataset of Section 5.2.1. We use the dataset to test the quality and reliability of the prediction model.

5.4.2. Measurement

We regard MSE, MAE and R^2 as the evaluation measures of prediction model. Mean Squared Error (MSE) is discribed in Formula 10.

Mean Absolute Error (MAE) is an average of the absolute difference between the actual observations and those predicted:

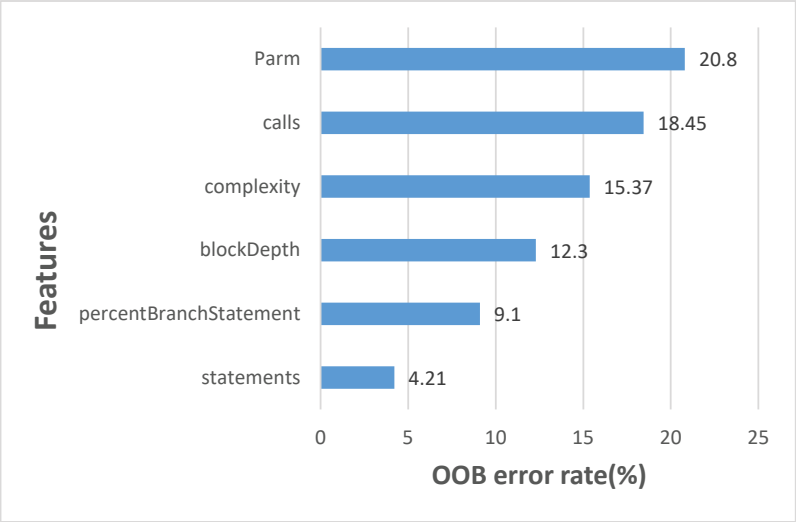


Figure 11: Measuring the features importance in predicating Edatasize using RF regression. The model developed using 3 mtry and 50 ntree. Higher %OOB error indicates greater importance.

$$MAE = \frac{1}{N}\sum_{t=1}^{N}|observed_t - predicted_t| \tag{11}$$

R-Squared is a more intuitive index to evaluate model, which is between 0 and 1:

$$R^2 = 1 - \frac{\sum(observed_t - predicted_t)^2}{\sum(observed_t - mean_t)^2} \tag{12}$$

In the above equations, N denotes the number of samples in test set. t denotes the current sample. $observed_t$ denotes the value of actual observations and $predicted_t$ denotes the value of predicted. $mean_t$ denotes the mean value.

Intuitively, the smaller MSE and MAE indicate the prediction model is more accurate. An acceptable value of R^2 is greater than 0.5 [36]. We use 10-fold cross validation to ensure the reliability of our results.

5.4.3. Results

It is worth to note that the MSE generally decreased while all of predictors (blockDepth, percentBranchStatement,complexity, statements,calls) were taken into account to train the model for `Etime`. The entire features produced lowest MSE using 10-fold cross validation (289.54 ms) as shown in Figure 12. Here, we select the features with the backward elimination search function.

Figure 12 shows that the MSE is 289.54 ms. And the MAE is 50.307ms, which we can regard as a small error when the response time of application is more than 20,000ms. R-Squared, as an intuitive index to evaluate model, is 0.786 in this paper.

In summary, the small error radio shows that our predication model can predict the execution costs of the remaining methods accurately. Based on this, AndroidOff can synthesize an offloading decision that minus the overall execution cost for application.

5.5. Comparison with Other Classifiers

5.5.1. The Setting

To show the effectiveness of Random Forest Regression, we use the same variables (blockDepth (B), percentBranchStatement (PE), complexity (CO), statements (S), calls (CA), Parm = int,double,String,int[],double[] (PA) of method.) and data items(Table 5) to train the SVM regression model and the Boosting regression model, and compare their accuracies.

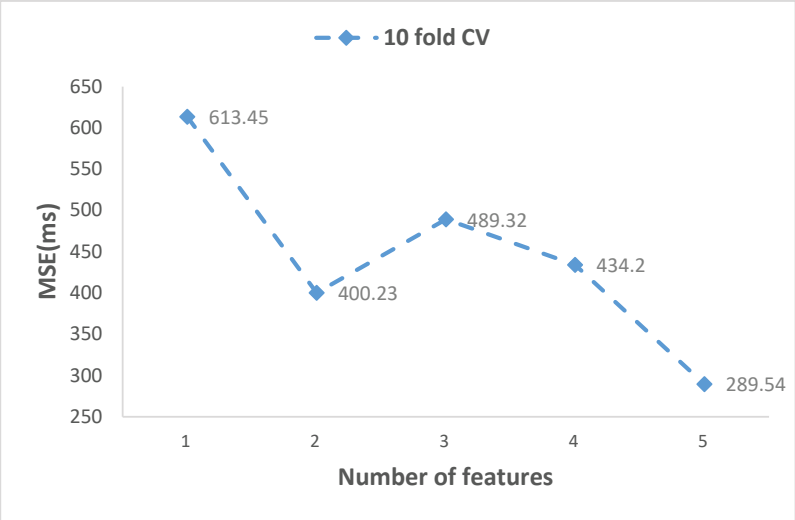


Figure 12: Selecting the optimal number of features using backward elimination search function. The MSE is calculated from the dataset(n=94) using 10-fold cross validation.

Table 6: Evaluation index of the three regression models for train set and test

model	subset	MAE	R^2
SVM	train set	29.743ms	0.841
	test set	59.552ms	0.737
Boosting	train set	52.660ms	0.764
	test set	66.529ms	0.715
Random Forest	train set	18.965ms	0.934
	test set	50.307ms	0.786

5.5.2. Measurement

Mean absolute error (MAE) and R-Squared (R^2) are used to evaluate prediction accuracy. MAE is a comprehensive measure of the deviation between the observed and predicted values of a sample. R^2 is commonly used to evaluate the quality of regression models. They are calculated according to Formula 11 and 12. Some research [65, 30, 56] shows that the smaller MAE indicates models fitting degree is better. For R-squared, the closer to 1 the better model is. We also use 10-fold cross validation to compare the results.

5.5.3. Result

The evaluation index of the three regression models for train set and test set are shown in Table 6. The value of MAE is sorted as $Boosting > SVM > RandomForest$ while the value of R^2 is sorted as $RandomForest > SVM > Boosting$. The optimal regression model yields the lowest MAE and highest R^2, hence Random Forest performs best in predicting execution costs.

5.6. Threat to validity

The threat to internal validity includes the measurement of response time and energy consumption. To make a fair comparison, we execute tasks for many times and calculate averages for comparison. Zhang [82] shows that PowerTutor [5] provides accurate, real-time power consumption estimates for power-intensive target application. In addition, the correlation between features and execution costs of methods in RQ3 may cause the threat to internal validity. However, our results show that MSE generally decreased while all of features were used to train the model

for execution costs. We should use a larger dataset to select more relevant features so as to improve the accuracy of model and reduce the threat.

The threat to external validity includes that selected Android application and network behavior in RQ1 may not fully reflect all real-world environment. The threat to construct validity include our network behavior. In our experiment, we set up a network environment to simulate the real-world environment as closely as possible. For instance, the two mobile devices separately represent low-end and high-end devices, and the network conditions between the mobile devices and the remote servers are different in different locations. Results exactly show the effectiveness of our approach. The differences of our simulated environment from the real-world environment are 1) the application is running in a single-user environment, so that the execution time for each invocation of the same class method on the same computation node is usually close to their average; and 2) the mobility model of mobile devices is not complex, so that the network conditions between the mobile devices and the same remote server in the same location are usually close to their average as well. Therefore, our framework can still work in the real-world environment but the performance improvement may be slightly different. However, this paper mainly focuses on determining which parts shall be offloaded in mobile edge computing, and the two issues above are orthogonal to the problem in this paper. Some related approaches can be introduced to reduce this threat, such as supporting multi-user cases via game-theoretic model [21, 23] and supporting complex mobility models via other offloading decision algorithms [72, 46]. The threat to external validity also includes the selected calibration dataset in RQ3. The calibration dataset was randomly split from dataset. The threat could be further reduced by training the prediction model many times.

6. Related Work

6.1. Offloading Mechanism

Offloading is a technique which aims at mitigating limitations of energy or QoS by delegating the execution of certain application software to remote resource rich infrastructures [80] like Clouds. Mobile Cloud Computing (MCC) is a promising approach to improve the performance and the battery consumption of mobile devices by previous offloading mechanisms, which can be classified into two broad categories [9] such as frameworks based on virtual machine cloning and frameworks based on code offloading. The former frameworks suspend and transfer mobile executions to VM clones on the cloud, and the latter frameworks offload intensive application components by invoking a remote procedure call (RPC) with annotations, special compilations, or binary modification. Some works [60, 38, 85, 25, 32] are VM/phone clone migration, which needs the same hardware platform as the server to retain a working synchronized image. Paranoid android [60] uses QEMU to perform attack detection on a remote server in the cloud where the execution of the software on the phone is mirrored in a virtual machine with minimal impact on phone performance and battery life. Virtual smartphone [38] uses Android x86 port to create virtual smartphone images in the mobile cloud efficiently on VMWare ESXi virtualization platform and requires developers to program applications that can interact and utilize the full power of the cloud resources. Phone mirroring [85] proposed a distinct augmentation framework which keeps a mirror for each smartphone on a computing infrastructure in the telecom network by providing different types of service, including computation offloading. CloneCloud [25] proposes cloud-augmented execution using a cloned VM image as a powerful virtual device and the application partition is offloaded as from a mobile device to the clone in the cloud. COMET [32] allows threads to migrate freely between machines depending on ALVMs that require massive modifications for different hardware platforms. Some works [27, 35, 75, 42, 61] are code migration, which should annotate code or follow specific programming models. In the MAUI system [27], parts of the application code with `[Remoteable]` annotation are offloaded to the server. And offloading problem is modeled to provide an optimal partitioning at runtime. Saab et al [35] replicates the application binary or source code at the server with a dynamic minimum-cut algorithm and an FSP-based protocol to transform the theory of mobile application partitioning into a practical solution. In Phone2Cloud [75], the remote execution manger gets required input data, it executes offloading computation on the cloud server, and sends back results to the offloading proxy. Cuckoo [42] makes partition based on the existing activity model in Android, and the framework receives method calls and evaluates whether to offload the method using heuristics information. The same as our work, Jade [61] checks the application and device status by monitoring the communication costs, work load variation, and energy status. Then an application is partitioned at the class level but a class must implement one of two interfaces to be offloadable. However, our work based on DPartner [84], which can transfer less granular object-threads from a mobile

device to a server by automatically refactoring Android code instead of abstracting massive threads. In addition, it can enable the on-demand offloading for an given Android app in a MCC as well as Mobile Edge Computing(MEC).

6.2. Offloading Strategy

Due to the resource heterogeneity of mobile devices and cloud services, the complexity of mobile applications and the characteristic of transferring a large amount of data, previous works focus on what computational tasks and data(what) at what place(where) to offload(*i.e.*, offloading strategy). To copy with what to offload, Ma et al[49] and Xian et al[76] do no partition for application and the entire application is either executed on the mobile system or offloaded. Some state-of-the-art computational offloading frameworks [8] employ adaptive algorithm in which elastic mobile applications dynamically distribute intensive partitions to the cloud server nodes. Ou et al[57] proposes an adaptive partitioning algorithm that partitions a given application into 1 unoffloadable partition and k offloadable partitions statically. While the result of [16], a dynamic partitioning, is the joint dynamic allocation of radio resources and offload scheduling. To deal with where to offload, some works[73],[74] combine the methods of analytic hierarchy process (AHP) and fuzzy technique to do offloading decision in cloud computing. MAUI[27] reduces latency by using a single-hop network and potentially saves battery by using WiFi or short-range radio, which belongs to cloudlet-based decision making. Ma et al[49] proposes a Cloud Assisted Mobile Edge computing (CAME) framework to optimize the usage of cloud resources and balance the workloads between the cloud and the mobile edge. According to Akherfi et al[9], the strategies can be classified as static and dynamic. In Phone2Cloud [75], the offloading decision is based on delay-tolerance threshold of user, which belongs to static strategy. In this paper, ANDROIDOFF implements an offloading decision by dynamically and automatically determining which portions of the application should be offloaded to the cloud, mobile edges or mobile device. Compared to traditional partitions, most of which are based on high-level abstractions of programs rather than actual applications, offloading at the granularity of objects can be applied directly and flexibly to the mobile offloading systems. In addition, ANDROIDOFF belongs to hybrid offloading decision-making, which determines where to perform each application task (locally, cloud or cloudlet) such that the energy consumption is minimized with a low delay penalty.

6.3. Code Analysis

Code analysis techniques are mostly used for application security [63, 67, 86] and bug detection [29, 81]. In the field of application security, [11] shows that a 17% cost reduction for reported security bugs is possible by using a static analysis tool. [12] shows that developers can correctly identify a warning from a static analysis tool(SAT) as a security threat that needs to be corrected. In terms of bug detection, [12] also shows that SAT is an effective early fault or vulnerability detector. SymbexNet[66] uses a combination of code analysis and symbol execution to automatically generate high coverage test packages in a rule-based network protocol specification and finds semantic errors in network protocol implementation. [10] shows that FindBugs, a static-analysis tool, can look for coding defects with fairly trivial analysis techniques in Java programs. While in this paper, code analysis techniques are used to analyze code in Android applications for cost estimation, which is not combined with offloading in previous work. Code analysis is applied in two aspects. First, ANDROIDOFF gets the call relations between methods and generate WCG based on Soot [69], a static code analysis tool. Second, ANDROIDOFF combine static analysis and running program dynamically to do cost estimation. However, most of previous works are only based on high-level program abstractions to build adaptive offloading decision algorithms.

7. Conclusion And Future Work

Although MCC and MEC have been introduced in general computation offloading to extend computing capability and battery capacity of mobile devices, we argue that it can fully release the potential of offloading if an application can determine the execution costs of its parts and which of parts shall be moved to MEC servers. To handle the problem, this paper proposes a novel approach, called ANDROIDOFF, that supports offloading at the granularity of objects of Android applications. For a given Android application, ANDROIDOFF first trains an estimation model to predict the execution costs of all methods through static analysis. Furthermore, based on the predicted cost, ANDROIDOFF synthesizes a decision model that minus the overall execution cost and determines which parts shall be moved to MEC

serves. We evaluate ANDROIDOFF on a real-world application. Results show that ANDROIDOFF can significantly improve the performance and save energy consumption, and predicting model is proved to be accurate.

In this paper, we consider that the execution costs of the method is mainly related to blockDepth, percentBranchStatements, complexity, statements, calls, and external calls, which can predict the execution costs of the method in most cases. While in the case of recursive calls, it is no easy to obtain the average frequency of executions about the code block since it shall be available at runtime. Some related work has been done to handle such problems [64, 51], which can be used in future work.

In addition, we intend to consider the overhead incurred by offloading the application as next steps of our research. For the prediction, it belongs to offline estimation model. And we should take the comparison of more models and affection factors into account to improve accuracy in the future work. For the online decision, some algorithm, such as Greedy Algorithm [41] and Genetic Algorithm [31], can be used to reduce overhead. To our knowledge, it takes about one minute to determine the offloading decision for Genetic Algorithm, while it just takes milliseconds to determine for Greedy Algorithm. Considering the performance and overhead of two algorithms, they are suitable to work in different situations. For instance, mobile devices are relatively static when we use VRS application, and Genetic Algorithm is more suitable and can achieve better application performance. In contrast, mobile devices are relatively dynamic when we use LRS application, and we shall use Greedy Algorithm for less decision time. Besides, because the decision algorithm is orthogonal to our model, other related works can be introduced to enhance ANDROIDOFF , such as PSO algorithm [48].

Acknowledgement

This work is supported by the National Key R&D Program of China under Grant No. 2018YFB1004800, Natural Science Foundation of China under Grant Nos. 61672159 and 61572313, Guiding Project of Fujian Province under Grant No. 2018H0017, the Talent Program of Fujian Province for Distinguished Young Scholars in Higher Education.

Reference

[1] Android application requirements. https://www.netmite.com/android/mydroid/development/pdk/docs/sys/temrequirements.
[2] Apps drain battery power. https://www.droidforums.net/forum/-droid-razr-support/216454-battery-drain.html.
[3] Huawei Honor 6. https://droidchart.com/en/huawei/huawei-honor-6-play-mya-al10-3563.
[4] Huawei Honor STF-AL00. https://www.amazon.com/HUAWEI-Honor-STF-AL00-Kirin-Smartphone/dp/B0711RJ92L.
[5] PowerTutor. http://powertutor.org.
[6] Soot, a Java Optimization Framework 3.2.0-SNAPSHOT API. https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/.
[7] WLAN RTT. https://android.googlesource.com/platform/hardware/libhardware_legacy/+/master/include/hardware_legacy/rtt.h.
[8] E. Abebe and C. Ryan. Adaptive application offloading using distributed abstract class graphs in mobile environments. *Journal of Systems & Software*, 85(12):2755–2769, 2012.
[9] K. Akherfi, M. Gerndt, and H. Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics*, 14(2210-8327):1–16, 2018.
[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
[11] D. Baca, B. Carlsson, and L. Lundberg. Evaluating the cost reduction of static code analysis for software security. In *ACM Sigplan Workshop on Programming Languages and Analysis for Security*, pages 79–88, 2008.
[12] D. Baca, P. Kai, B. Carlsson, and L. Lundberg. Static code analysis to detect software security vulnerabilities - does experience matter? In *International Conference on Availability, Reliability and Security*, pages 804–810, 2009.
[13] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *Proc. ACM SIGOPS European workshop*, pages 87–92, 2002.
[14] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *Proc. MobiSys*, pages 272–285, 2007.
[15] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proc. MobiSys*, pages 273–286, 2003.
[16] S. Barbarossa, S. Sardellitti, and P. D. Lorenzo. Computation offloading for mobile cloud computing based on wide cross-layer optimization. In *Future Network and Mobile Summit*, pages 1–10, 2013.
[17] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *IEEE INFOCOM*, pages 1285–1293, 2013.

[18] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Acm Sigplan International Workshop on State of the Art in Java Program Analysis*, pages 27–38, 2012.
[19] M. E. Berglund, J. Duvall, and L. E. Dunne. A survey of the historical scope and current trends of wearable technology applications. In *Proc. ISWC*, pages 40–43, 2016.
[20] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
[21] X. Chen. Decentralized computation offloading game for mobile cloud computing. *Parallel & Distributed Systems IEEE Transactions on*, 26(4):974–983, 2014.
[22] X. Chen, S. Chen, Y. Ma, B. Liu, Y. Zhang, and G. Huang. An adaptive offloading framework for android applications in mobile edge computing. *SCIENCE CHINA Information Sciences*, pages –, 2019.
[23] X. Chen, L. Jiao, W. Li, and X. Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808, 2016.
[24] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *Proc. ASE*, pages 429–440, 2015.
[25] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proc. EuroSys*, pages 301–314, 2011.
[26] A. Criminisi, D. Robertson, E. Konukoglu, J. Shotton, S. Pathak, S. White, and K. Siddiqui. Regression forests for efficient anatomy detection and localization in computed tomography scans. *Medical Image Analysis*, 17(8):1293–1303, 2013.
[27] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. MobiSys*, pages 49–62, 2010.
[28] F. E, H. M. L, H. G, and W. IH. Data mining in bioinformatics using weka. *Bioinformatics*, 20(15):2479–2481, 2004.
[29] F. Ebert, F. Castor, and A. Serebrenik. An exploratory study on exception handling bugs in java programs. *The Journal of Systems & Software*, 106(C):82–101, 2015.
[30] G. R. Finnie, G. Wittig, and J.-M. Desharnais. A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models. *Journal of Systems & Software*, 39(3):281–289, 1997.
[31] D. E. Goldberg. Genetic algorithms in search, optimization and machine learning. 1989.
[32] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: code offload by migrating execution transparently. In *Usenix Conference on Operating Systems Design and Implementation*, pages 93–106, 2012.
[33] S. Goyal and J. Carter. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proc. WMCSA*, pages 186–195, 2004.
[34] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proc. PerCom*, page 107, 2003.
[35] Saab, Salwa Adriana and Saab, Farah and Kayssi, Ayman and Chehab, Ali and Elhajj, Imad H. Partial mobile application offloading to the cloud for energy-efficiency with security measures. *Sustainable Computing Informatics & Systems*, 8:38–46, 2015],.
[36] W. S. Humphrey. Toward a discipline for software engineering. In *Sei Conference on Software Engineering Education*, 1992.
[37] G. C. Hunt, M. L. Scott, et al. The Coign automatic distributed partitioning system. In *Proc. OSDI*, pages 187–200, 1999.
[38] M. Itoh, Eric Y. Chen. Virtual smartphone over ip. 2010.
[39] J. K. Jaiswal and R. Samikannu. Application of random forest algorithm on feature subset selection and classification and regression. In *Computing and Communication Technologies*, page 10, 2017.
[40] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. 2013.
[41] Q. Kang, H. He, and H. Song. Task assignment in heterogeneous computing systems using an effective iterated greedy algorithm. *Journal of Systems & Software*, 84(6):985–992, 2011.
[42] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Proc. MobiSys*, pages 59–79. Springer, 2010.
[43] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. INFOCOM*, pages 945–953, 2012.
[44] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.
[45] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.
[46] T. Lei, S. Wang, J. Li, and F. Yang. Aom: adaptive mobile data traffic offloading for m2m networks. *Personal & Ubiquitous Computing*, 20(6):1–11, 2016.
[47] P. Li, X. Yu, X. Peng, Z. Zheng, and Y. Zhang. Fault-tolerant cooperative control for multiple UAVs based on sliding mode techniques. *Science China Information Sciences*, 60(7):070204, 2017.
[48] B. Lin, F. Zhu, J. Zhang, J. Chen, X. Chen, N. Xiong, and J. Lloret. A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing. *IEEE Transactions on Industrial Informatics*, PP(99):1–1, 2019.
[49] X. Ma, S. Zhang, W. Li, P. Zhang, C. Lin, and X. Shen. Cost-efficient workload scheduling in cloud assisted mobile edge computing. In *Ieee/acm International Symposium on Quality of Service*, pages 1–10, 2017.
[50] S. E. Mahmoodi, R. Uma, and K. Subbalakshmi. Optimal joint scheduling and cloud offloading for mobile applications. *IEEE Transactions on Cloud Computing*, 2016.
[51] P. Maresca. *The Execution Time of an Algorithm: Advanced Considerations*. 2015.
[52] P. E. Mcknight and J. Najab. *Mann-Whitney U Test*. 2010.
[53] H. Mei and X. Liu. Software techniques for internet computing: Current situation and future trend. *Chinese Science Bulletin*, 55(31):3510–3516, 2010.
[54] O. Munoz, A. Pascual-Iserte, and J. Vidal. Optimization of radio and computational resources for energy efficiency in latency-constrained application offloading. *IEEE Transactions on Vehicular Technology*, 64(10):4738–4755, 2015.
[55] O. Mutanga, E. Adam, and M. A. Cho. High density biomass estimation for wetland vegetation using worldview-2 imagery and random

forest regression algorithm. *International Journal of Applied Earth Observations and Geoinformation*, 18(1):399–406, 2012.
[56] A. B. Nassif, D. Ho, and L. F. Capretz. Towards an early software estimation using log-linear regression and a multilayer perceptron model. *Journal of Systems & Software*, 86(1):144–160, 2013.
[57] S. Ou, K. Yang, and A. Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *IEEE International Conference on Pervasive Computing and Communications*, pages 116–125, 2006.
[58] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005.
[59] M. Philippsen and M. Zenger. Javaparty–transparent remote objects in Java. *Concurrency: Practice and experience*, 9(11):1225–1242, 1997.
[60] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Computer Security Applications Conference*, pages 347–356, 2010.
[61] H. Qian and D. Andresen. Jade: Reducing energy consumption of android app. *International Journal of Networked & Distributed Computing*, 3(3):150–158, 2015.
[62] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel. Computation offloading and resource allocation for low-power IoT edge devices. In *Proc. WF-IoT*, pages 7–12, 2016.
[63] L. Sampaio and A. Garcia. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems & Software*, 113:337–361, 2016.
[64] C. Shan, Z. Yu, C. Hu, J. Xue, and L. Wu. Optimization of program recursive function calls analysis method. *Automatic Control & Computer Sciences*, 50(4):253–259, 2016.
[65] R. Silhavy, P. Silhavy, and Z. Prokopova. Analysis and selection of a regression model for the use case points method using a stepwise approach. *Journal of Systems & Software*, 125, 2016.
[66] J. S. Song, C. Cadar, and P. Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, 2014.
[67] J. Thom, L. K. Shar, D. Bianculli, and L. Briand. Security slicing for auditing common injection vulnerabilities. *Journal of Systems & Software*, 137:766 – 783, 2018.
[68] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili. Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges. *IEEE Communications Magazine*, 55(4):54–61, 2017.
[69] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *Proc. CASCON*, pages 214–224, 2010.
[70] Q. Wang, F. Reimeier, and K. Wolter. Efficient image stitching through mobile offloading. *Electronic Notes in Theoretical Computer Science*, 327:125–146, 2016.
[71] Q. Wang and K. Wolter. Reducing task completion time in mobile offloading systems through online adaptive local restart. 2015.
[72] S. Wang, T. Lei, L. Zhang, C.-H. Hsu, and F. Yang. Offloading mobile data traffic for qos-aware service provision in vehicular cyber-physical systems. *Future Generation Computer Systems*, 61(C):118–127, 2016.
[73] M. Whaiduzzaman, A. Gani, N. B. Anuar, M. Shiraz, M. N. Haque, and I. T. Haque. Cloud service selection using multicriteria decision analysis. *The Scientific World Journal,2014,(2014-2-13)*, 2014(9):459375, 2014.
[74] H. Wu, Q. Wang, and K. Wolter. *Optimal Cloud-path Selection in Mobile Cloud Offloading Systems Based on QoS Criteria*. IGI Global, 2013.
[75] F. Xia, F. Ding, J. Li, X. Kong, L. T. Yang, and J. Ma. Phone2cloud: exploiting computation offloading for energy saving on smartphones in mobile cloud computing. *Information Systems Frontiers*, 16(1):95–111, 2014.
[76] C. Xian, Y. H. Lu, and Z. Li. *Adaptive computation offloading for energy conservation on battery-powered systems*. IEEE Computer Society, 2007.
[77] F. Yang, J. Li, T. Lei, and S. Wang. Architecture and key technologies for internet of vehicles: a survey. *Journal of Communications and Information Networks*, 2(2):1–17, 2017.
[78] K. Yang, S. Ou, and H.-H. Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *IEEE communications magazine*, 46(1), 2008.
[79] Ying, Ying, Wang, and Han. Dynamic random regression forests for real-time head pose estimation. *Machine Vision & Applications*, 24(8):1705–1719, 2013.
[80] A. Yousafzai, A. Gani, R. M. Noor, A. Naveed, R. W. Ahmad, and V. Chang. Computational offloading mechanism for native and android runtime based mobile applications. *Journal of Systems & Software*, 121(C):28–39, 2016.
[81] H. Zhang, H. B. K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang, and H. Mei. Checking enforcement of integrity constraints in database applications based on code patterns. *Journal of Systems & Software*, 84(12):2253–2264, 2011.
[82] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, and Y. Lei. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *IEEE/ACM/IFIP International Conference on Hardware/software Codesign & System Synthesis*, 2010.
[83] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Networks and Applications*, 16(3):270–284, 2011.
[84] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. *OOPSLA*, 47(10):233–248, 2012.
[85] Zhao, Caixia, Sencun, and Guohong. Mirroring smartphones for good a feasibility study. *Zte Communications*, 09(1):9–14, 2011.
[86] X. Zhou, K. Wu, H. Cai, S. Lou, Y. Zhang, and G. Huang. Logpruner: detect, analyze and prune logging calls in android apps. *Science China Information Sciences*, 61(5):050107, 2018.

Appendix

Table 7 shows the WCGs that were extracted by ANDROIDOFF. Its columns list the callers, the callees, and weight of each method call. ANDROIDOFF extracts features for each method in subject application in Definition 2.

Table 7: Call relations between methods

R	Caller	CallerMethodName	Callee	CalleeMethodName	Weight
$r_{1,4}$	m_1	onCreate	m_4	patrol	1
$r_{2,6}$	m_2	capture	m_6	takePhoto	1
$r_{2,5}$	m_2	capture	m_5	imageProcess	1
$r_{2,11}$	m_2	capture	m_{11}	store	1
$r_{4,12}$	m_4	patrol	m_{12}	down	1
$r_{4,13}$	m_4	patrol	m_{13}	up	1
$r_{4,14}$	m_4	patrol	m_{14}	left	1
$r_{2,15}$	m_4	patrol	m_{15}	right	1
$r_{4,16}$	m_4	patrol	m_{16}	speedUp	1
$r_{4,17}$	m_4	patrol	m_{17}	speedDown	1
$r_{4,18}$	m_4	patrol	m_{18}	A^*	1
$r_{4,3}$	m_4	patrol	m_3	getMap	1
$r_{5,2}$	m_5	patrol	m_2	capture	5
$r_{5,7}$	m_5	imageProcess	m_7	grayProcess	1
$r_{5,8}$	m_5	imageProcess	m_8	resize	1
$r_{5,9}$	m_5	imageProcess	m_9	getTarget	1
$r_{5,10}$	m_5	imageProcess	m_{10}	ocrAnalysis	1

Table 8: Offloading decision

Device	MovableObject	Location			
		Garden	playground	TeachingBuilding	LaboratoryBuilding
Mobile1	pathPlanning	Local	Local	Local	Edge2
	imageProcessing	Local	Edge1	Edge2	Cloud
	grayProcess	Local	Edge1	Edge2	Cloud
	ocrProcess	Local	Edge1	Edge2	Cloud
	store	Local	Local	Local	Local
Mobile2	pathPlanning	Local	Local	Local	Local
	imageProcessing	Local	Local	Edge2	Cloud
	grayProcess	Local	Local	Edge2	Cloud
	ocrProcess	Local	Local	Edge2	Cloud
	Store	Local	Local	Local	Local

Table 8 shows the offloading plan of ANDROIDOFF. The results show that the effectiveness of ANDROIDOFF is close to that of the ideal approach. In addition, the application can also completes all tasks and runs as normal with this deployment. So our plan is correct. The Logger depends a lot on network conditions. Particularly, overall offloading can not weight transport latency against execution time. The ideal approach is our target. It means we hope the result of our plan calculated in theory and the ideal approach executed in practical is as close as possible.