Elicit: Efficiently Identify Computation-intensive Tasks in Mobile Applications for Offloading

Mohammed A. Hassan, Qi Wei and Songqing Chen

mohammeh@netapp.com, NetApp Inc.
{qwei2,sqchen}@gmu.edu, Department of Computer Science, George Mason University

***Abstract*—As mobile devices are battery powered and have less computing resources, plenty of research has been conducted on how to efficiently offload computing-intensive tasks in a mobile application to more powerful counterpart. However, prior research either implicitly assumes that the computing-intensive tasks are known in advance or the application developers will make special notations about them.**

In this paper, we design a framework `Elicit` to efficiently identify the computation-intensive tasks in mobile applications for offloading. Furthermore, we also consider the response time savings dynamically when deciding whether to offload a task based on the runtime system resources. A prototype of `Elicit` is built based on the Dalvik VM. Our evaluation with some popular Android applications from Google Play shows that `Elicit` can efficiently find an application's computing-intensive task and save response time and energy consumption when these tasks are offloaded.

I. Introduction

The increasing popularity of the mobile devices is attracting more and more developers to develop resource intensive applications for mobile devices. But mobile devices are constrained by limited computing power and battery supply. Thus, a lot of efforts have been made to augment mobile devices' capability with servers or clouds. For example, prior research [12], [22], [13], [21], [18] has proposed to partition the application and offload computation-intensive portions to more powerful counterparts such as servers [13], [18], while application-specific offloading is considered in [21]. In practice, the Amazon SILK browser [1] splits the browsing tasks between the server and the user hand-held devices.

However, to implement the offloading, these models either require the application developers to provide special notation on the resource intensive methods for offloading, or implicitly assume that the resource intensive methods are already known [13], [18]. Clone-based approaches [12] can offload applications without modifications to applications, but they require a full mobile image running on the cloud, which brings high synchronization overhead [14], [22]. Such requirements prevent these models from being deployed in practice.

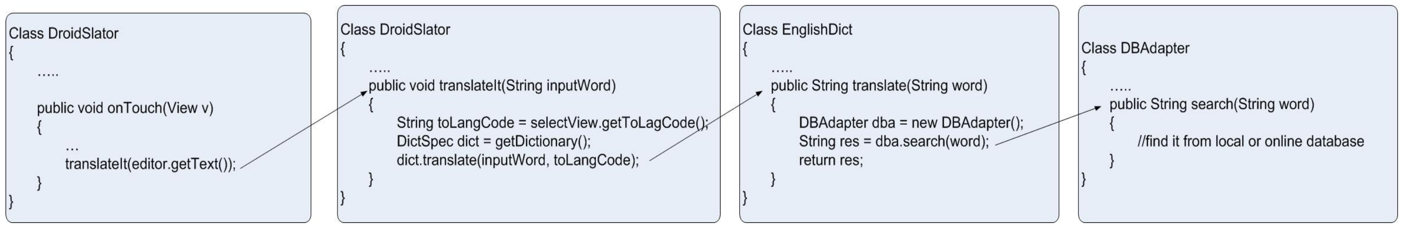
In addition, Young et al. [18] suggested to offload the computation-intensive methods to enhance the performance of mobile applications when the size of the method parameters is greater than 6MB. In practice, computation-intensive methods do not necessarily have direct relationship with the size of input data. Moreover, offloading the resource-intensive methods may not save energy and response time in every circumstance. MAUI [13] adopts 0-1 integer linear programming (ILP) to solve the problem of offloading to maximize performance gain in terms of energy or response time. However, in practice not every method can be offloaded. Some methods may access to camera or other sensors of the mobile device, which can not be offloaded and have to be executed on the mobile device. While choosing the methods to offload, one has to consider these constraints which are not considered in previous research. Previous research [13], [22], [10] mainly relies on the developers for ensuring that these methods are not constrained, which is not practical either.

To address this problem, in this paper, we propose to build a framework, called `Elicit`, to **E**fficient**L**y **I**dentify **C**omputation-**I**ntensive **T**asks in mobile applications for offloading. Considering the fact that even a very simple mobile application can consist of hundreds of different method calls, `Elicit` works by first profiling mobile applications to find the methods' resource consumption, their dependency on each-other, and constraints which may hinder them from being offloaded. Moreover, if the server is on the cloud, the resources are abundant, but the network delay could be large and more dynamic. If the server is nearby (such as Fog computing [11]), the resources may be less powerful, but the network latency is also smaller. Then, considering the environmental setup at runtime (e.g., network latency and bandwidth between the mobile device and the server, size of the overhead data, etc.), `Elicit` dynamically chooses an optimal partition of the application for offloading based on the projected gains.

An `Elicit` prototype is implemented on Android Dalvik virtual machine. We evaluate it with a few popular Android applications. The results show that `Elicit` can find the most resource consuming methods of the applications for offloading, which are consistent with previous studies [18], [15]. Our experimental results also confirm that `Elicit` can dynamically choose the optimal partition of applications for offloading the method to improve the response time.

While details are provided in the later context, the main contributions of this paper include:

- `Elicit` can efficiently find the candidate methods that are not constrained and appropriate for offloading.
- `Elicit` dynamically chooses the optimal partition of mobile applications to offload computation-intensive methods to improve the response time.
- A prototype of `Elicit` is implemented in the Android Dalvik virtual machine and evaluated with a few popular mobile applications from Google Play to demonstrate its effectiveness.



```
Class DroidSlator
{
    .....
    public void onTouch(View v)
    {
        ...
        translateIt(editor.getText());
    }
}
```

```
Class DroidSlator
{
    .....
    public void translateIt(String inputWord)
    {
        String toLangCode = selectView.getToLagCode();
        DictSpec dict = getDictionary();
        dict.translate(inputWord, toLangCode);
    }
}
```

```
Class EnglishDict
{
    .....
    public String translate(String word)
    {
        DBAdapter dba = new DBAdapter();
        String res = dba.search(word);
        return res;
    }
}
```

```
Class DBAdapter
{
    .....
    public String search(String word)
    {
        //find it from local or online database
    }
}
```

Fig. 1: Call-flow of DroidSlator Application

II. Motivations

For any existing mobile application, one has to identify the most appropriate resource-consuming task before any of the existing offloading mechanisms can be used to offload that. In addition, these tasks should be unconstrained so that they can be executed on the server. Existing research on mobile application offloading however commonly ignores this problem.

Previous studies either manually identify the methods for offloading [15], [13], [18] or use more fine grained offloading mechanism to offload threads [14]. It is possible to find the resource intensive method of an application manually to achieve the optimal performance gain. But the main drawbacks are (1) it requires the programmer's knowledge to find the appropriate method to offload; (2) the cost is prohibitively high to offload an existing application in this way, given that there are hundreds of methods in an even very simple application.

On the other hand, fine grain offloading can provide seamless thread migration without any modification of the source code and it also does not require any special user notation for offloading. But the thread migration has two major limitations. The first is that when migrating a thread, it requires a clone of the mobile OS image to be saved on the server side for executing the offloaded thread there. While offloading the thread, these two images must be synchronized continuously, which brings a lot of network overhead. For example, the overhead to offload a single method in Cloudlet [22] is around 100MB. The second is that while VM based cloning is flexible for the application to be offloaded at any point during the execution, it does not find the appropriate resource intensive computation to offload. For example, COMET [14] migrates a thread only when its on-device execution time exceeds twice of the round trip time from the mobile device to the server (and later this threshold is changed to the average synchronization overhead). In this way, the offloading performance gain is not guaranteed: it is possible that the thread migration overhead may be greater than the performance gain obtained by offloading the rest of the execution of the thread.

Zhang et al. [24] showed that the performance gain can be achieved by distributing an application execution between the mobile device and the server by adopting a shortest path algorithm to find an optimal cut to minimize the offloading overhead. Although this solution provides a generic approach for application partitioning, it does not consider many practical constraints for partitioning real-world applications. For example, it is possible to find a partition that gives the best performance gain, but at the same time the methods (in the optimal partition) can not be offloaded due to some practical constraints, such as the candidate methods may need to access the mobile device's camera or other I/O devices. In addition, this work does not provide any guideline on how to profile the applications to find the bottleneck to offload.

Figure 1 shows an example of how the most time-consuming methods of the DroidSlator application [3] (Android dictionary app) are invoked. Here the `onTouch()` method consumes most of the execution time of the application. It calls the `translatrIt()` method. Thus, the `onTouch()` method's execution time includes the execution time of `translateIt()` as well. In this way, `translateIt()` also includes the execution time of `translate()` and subsequently that of the `search()` method.

This example also shows that it is not possible to offload all the methods. For example, the `onTouch()` method interacts with the `View Display`, so it is not feasible to offload it. The `translateIt()` method interacts with the user input so it is not possible to offload it either. Two other remaining candidate methods are `translate()` and `search()` and, we want to offload the one with less overhead compared to the offloading gain.

Once the list of candidate methods for offloading is identified, the next challenge is to choose the most appropriate methods to offload, at runtime. Our prior research [15] has shown that such a decision should be dynamic, depending on the execution environment (fog or cloud) and sometimes it may be worth executing the method on the device rather than to offload (to the cloud). Moreover, if we have multiple candidate methods to offload, we should offload the method(s) which provide(s) the best performance gain considering the execution environment (fog or cloud).

III. Design of Elicit

To improve the response time, we need to partition the mobile application to offload the most resource consuming methods. These methods should be eligible for offloading, i.e. the method should not access any other methods or variables that are related to the sensors, i.e., camera, user I/O etc. of the mobile device. Whenever a method is offloaded to the server, the methods upon which it is dependent are offloaded as well. Moreover, while offloading, we also need to send all the global or class variables that are accessed by the offloaded methods or its descendants. Otherwise the methods have to contact the mobile device again to access these variables, which we want to minimize.

With these constraints and objectives, we first formulate the problem as a constrained graph partition problem, then we aim to find the optimal solution for `Elicit`.

A. Objective and Constraints

We can formulate the candidate methods finding as a constrained graph partitioning problem. Here we have a list of methods M, and each method $m \in M$ has the following members: i) execution cost on the mobile device, denoted as m_m, ii) offloading (the parameters) and execution cost on server, denoted as m_s, iii) class of this method: MC (Method Class), iv) list of global or class variables accessed by the method, denoted as V, v) list of the classes of V of this method, denoted as VC (Variable Class), vi) list of all the variables reachable from this method, denoted as RV (Reachable Variables), vii) list of all the methods reachable from this method, denoted as RM (Reachable Methods), viii) list of the classes of the methods in RM, denoted as MCC (Method Closure Class), and ix) list of the classes of all the variables of the methods in RM, denoted as VCC (Variable Closure Class).

Let us also denote the list of classes that can not be offloaded as CC (Constrained Class), and the method call graph as MCG. Each node of MCG represents a method and an edge e_{xy} represents whether a method x is invoking another method y. Assume that the execution cost of a method m in the mobile device is m_m and on the server (without considering the parameters sending cost) m_{swp}. In addition, the cost of invoking a method m from method x is denoted as e_{xm}, and the cost for accessing a variable (global or class) v is e_{mv}.

So the cost of executing a method on the server (including the overhead) is:

$$m_s = m_{swp} + \sum_{\forall x \in \{M-m\}} e_{xm} + \sum_{\forall x \in RV} e_{mx}. \tag{1}$$

And the cost saved by offloading a method is

$$b_m = m_m - m_s. \tag{2}$$

Thus, our goal is to partition the application's MCG into two partitions S and T and offload all the methods $m \in S$ to the server side such that:

$$Execution\ Time = \sum_{\forall m \in T} m_m + \sum_{\forall m \in S} m_s \tag{3}$$

is minimized, subject to the following constraints:

$$\forall m \in S\ \forall c \in m \cdot MCC \rightarrow c \notin CC \tag{4}$$

$$\forall m \in S\ \forall c \in m \cdot VCC \rightarrow c \notin CC \tag{5}$$

These constraints ensure that no method or variable accessed by the offloaded method(s) belongs to the constrained class set. Note that here the cost of executing a method on the server may change based on the environment, i.e. the communication cost may be low but the execution cost might be high (in fog) or vice-versa (in cloud). Such costs dynamically impact the offloading decision.

Algorithm 1 Find Reachable Methods()

```
1: for $\forall m \in M$ do
2:     $m \cdot RM \leftarrow DFS(m)$
3: end for
4: Begin Function{DFS}{$m$}: $list$
5:     $list \cdot add \leftarrow m \cdot MC$
6:     if $m$ is visited then
7:         do nothing
8:     else
9:         for $\forall x \in neighbours\ of\ m$ do
10:            $list \cdot add \leftarrow DFS(x)$
11:        end for
12:    end if
13:    $m$ is visited
14:    return list
15: End Function
```

Algorithm 2 Find Reachable Variables(m)

```
1: $list \cdot add \leftarrow empty$
2: for $\forall x \in m \cdot RM$ do
3:     if $list$ doesn't contain $x \cdot V$ then
4:         $list \cdot add \leftarrow x \cdot V$
5:     end if
6: end for
7: $m \cdot RV \leftarrow list$
```

B. Proposed Solution

With the problem formulated and metrics defined, we map our problem of dynamic application partitioning (based on environments and constraints) to the `Project Selection Problem` [16]. Kleinberg et al. [16] proposed this `Project Selection Algorithm` for profit maximization. Suppose there is a set of projects which provide profit, and a set of equipments which require some cost to purchase. Each project is dependent on one or more of the equipments. Kleinberg et al. [16] have shown that the max flow min-cut theorem can give an optimal partition to find the list of projects and equipments to be chosen to maximize the profit. Kleinberg et al. proposed their method for a dependency graph as well,

Algorithm 3 Find Method Closure Class(m)

```
1: $list \cdot add \leftarrow empty$
2: for $\forall x \in m \cdot RM$ do
3:     if $list$ doesn't contain $x \cdot VC$ then
4:         $list \cdot add \leftarrow x \cdot VC$
5:     end if
6: end for
7: $m \cdot MCC \leftarrow list$
```

Algorithm 4 Find Variable Closure Class(m)

```
1: $list \cdot add \leftarrow empty$
2: for $\forall x \in m \cdot RM$ do
3:     if $list$ doesn't contain $x \cdot MC$ then
4:         $list \cdot add \leftarrow x \cdot MC$
5:     end if
6: end for
7: $m \cdot VCC \leftarrow list$
```

Algorithm 5 Candidate Method Selection Algorithm()

```
1: for $\forall m \in M$ do
2:     find the members of $m$ from Algorithm 1, 2, 3, and 4
3: end for
4: for $\forall m \in M$ do
5:     if $m \cdot MC \in CC$ then
6:         discard $m$
7:     else if $\exists x \in m \cdot MCC$ and $x \in CC$ then
8:         discard $m$
9:     else if $\exists x \in m \cdot VCC$ and $x \in CC$ then
10:        discard $m$
11:    else
12:        $CAN \cdot add\ m$
13:    end if
14: end for
15: return $CAN$
```

Algorithm 6 Offloading Algorithm()

```
1: Train the two MLP learners with labeled data
2: $CAN \leftarrow$ List of methods can be offloaded obtained from
   Algorithm 5
3: for each $m \in CAN$ do
4:     $m_m \leftarrow Predicted\ OnDevice\ Execution\ Time$
5:     $m_s \leftarrow Predicted\ OnServer\ Execution\ Time$
6: end for
7: $O \leftarrow$ list of methods to be offloaded according to Project
   Selection Algorithm [16]
8: if $O$ is empty then
9:     execute the app locally
10:    monitor the OnDevice Execution Time and
11:    train the OnDevice Execution Time learner with the new
       data
12: else
13:    offload the methods $m \in O$
14:    monitor the OnServer Execution Time and
15:    train the OnServer Execution Time learner with the new
       data
16: end if
```

where some project execution depends on some of the other projects. Our goal is to find a similar set of methods for offloading to minimize the cost by optimal partitioning. But in our set of methods, we may have many methods that can not be offloaded due to the previously mentioned constraints. So, we have to filter out the constrained methods first, only after that we may map the problem to Kleinberg's method.

To filter out the constrained methods, at first we find the methods' parameters mentioned in the previous subsection III-A. Given the execution time m_m and m_s, class MC, the list of global or class variables V of the method (and their class VC), and the list of caller callee methods (details of how these information can be found is stated in implementation section IV), we build up the method call graph MCG from the list of the caller and callee methods. To find the list of the reachable methods RM, we propose Algorithm 1 which is a variation of Depth First Search Algorithm [23]. Once we find RM from Algorithm 1, we find the RV, MCC, and VCC according to Algorithm 2, 3, and 4, respectively. From these parameters, we discard the methods that violate the constraints 4 and 5. Thus, Algorithm 5 finds CAN the list of methods eligible for offloading.

Once we get the list of the unconstrained methods CAN and their `Method Call Graph` MCG, we map our problem as follows. We convert this unconstrained MCG to `Method Selection Graph`. We introduce two nodes m and m' for each method $m \in CAN$ to the new graph. We also introduce two dummy nodes λ (source) and μ (sink). Each node m has an edge from λ and the edge has capacity m_m of that method, which represents the on-device execution time of the method. Each node m' (corresponding to the additional node introduced for each method) has an edge from itself to μ with capacity m_s, the method's execution time on the server side. Then we set the edges between m and m' with capacity $C = \sum_{m \in M}(m_m + m_s)$. We also set the internal edge's capacity to be C as well. In this way, we map our problem to `Project Selection Problem`.

Let there are P projects and Q equipments. Each project $p_i \in P$ results in some profit pr_i, and each equipment $q_i \in Q$ has some associated cost ct_i. Each project depends on one or more equipments. Here our goal is to select such projects so that the overall profit is maximized. At first, we build the `Project selection graph` where in addition to the projects and equipments, we introduce two additional nodes: the source and the sink. The capacity of the edges between the source and the projects are pr_i, while the capacity between the equipments and sink is ct_i. Thus we find the max-flow min-cut solution for this `Project Selection Graph` and select the project (to execute) and the associated equipments (to purchase) belongs to min-cut partition to maximize the profit.

In our method selection problem, the cost equals to the offloading overhead of the methods to the server side; while the profit is the time we save by offloading, which equals to the on-device execution time. As a result our cost is m_s and profit is m_m. After mapping our problem, we find the max-flow min-cut solution for this `Method Selection Graph` and select the methods to maximize the response time savings. We find the max-flow by iterating over the graph. Each time we find a path where we may add more content to the edges from the source to the sink respecting their capacities. Once the graph gets saturated after we achieved the max-flow, we find the min-cut by finding the edges whose capacities are not full. By choosing these nodes, we make sure that the profit is greater than the cost. When the capacity is not full in the final result, we know that the cost of executing that method on the server-side is less than the on-device execution cost (considering the offloading cost of the dependent methods to the server as well). In this way, we can optimally partition the application for offloading provided we can predict m_m and m_s accurately by our prediction models described in the next section III-C. The complexity for this solution is $O(VE^2)$, where V and E are the number of vertices and edges of the `Method Selection Graph`.

C. Response Time Prediction

Our model should predict the methods' execution time both on the server-side and on the mobile device and partition the application in an optimal way (based on Project Selection Problem [16]). A recent study [15] investigated different

TABLE I: Relative Absolute Error (%) of Different Learning Models

	Droid-Slator	Mat-Cal	Math-Droid	N-Queen	Picaso
LR	187.81	189.89	5.20	78.99	36.56
SVM	136.38	74.27	4.21	17.98	17.60
DT	149.23	132.81	46.56	105.90	75.22
MLP	49.18	60.55	6.06	16.71	17.05

models such as Threshold Based (TB), Naive Bayes (NB), Linear Regression (LR), Support Vector Machine (SVM), Decision Tree (DT), and Multilayer Perceptron (MLP) to make offloading decisions. Among these learning models, MLP, SVM, and DT are found to perform better than others.

We also consider these models in `Elicit`, excluding TB and NB as they cannot predict a specific output variable. To determine the most appropriate model, we have conducted some experiments with some mobile applications. Table III in section V briefly describes these applications as well as the methods' name for which we are predicting the on-server execution time to demonstrate different model's accuracy (section III-B shows how we can find these candidate methods automatically). We have five different configurations in the experiments to simulate different environments as mentioned in section V. In each configuration, the available resources, such as CPU, memory, data size along with the network parameters are dynamically changing.

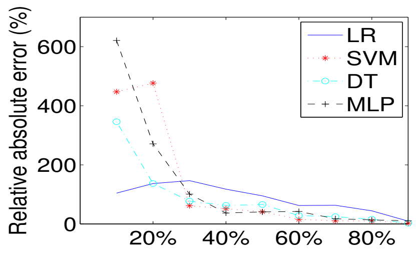


Fig. 2: Gradual Learning

We implement these models using open source library `Weka` [9] on the Android phone Google Nexus one with 512 MB memory and 1 GHz CPU. We have trained the models with 50% of the data (for each application) because more training could help some models. The data are from users' trace of different Android applications for a duration of one month collected by us. During the data collection, we have changed the memory and CPU availability on the server and used different network configurations. We have conducted 10 experiments for each application in each setting and we change the computation data size whenever possible for these applications. Figure 5 in section V shows the data size for all the 50 experiments in different environments. Table I shows the average mean error rate of these 50 experiments in total for each application. The results indicate that MLP, SVM, and DT have better accuracy (or less error) than LR.

We also evaluate their performance with online learning over the process. Figure 2 shows the accuracy of different learning model over the time for 50 experiments of MatCal [4]. The x-axis represents the percentage of the data we are using for training while the rest of them are test data, while the y-axis represents the relative absolute error percentage. We observe similar trends for the other applications, so we omit them for brevity. Figure 2 shows that LR performs better while being trained with a small percentage of data, but when the training set grows greater than 30%, SVM and MLP start to perform significantly better than LR, reflecting their online learning capability.

TABLE II: Overhead of Different Classifiers

	Classification		Training (total 150 instances)	
	Time (ms)	Energy (mJ)	Time (ms)	Energy (mJ)
MLP	8.76	8.33	8290	10000
LR	9.56	4.66	604	1100
SVM	6.82	5.00	815	1500
DT	10.26	3.67	454	1100

Table II further shows the classification overhead of a single experiment and the one time training overhead for different models. Note that although DT is also lightweight with moderate accuracy, we exclude it first since it cannot extrapolate its decision and suffers from over-fitting. Both MLP and SVM perform well in terms of accuracy and prediction time, can capture the relationship between multiple features, and support online training [19]. SVM is easy to train and free from over-fitting. Although MLP takes a bit more time for training [15], it supports on-line learning and works well in noisy environments, and the amortized cost for one experiment is low. Thus `Elicit` can use either of them, and we will evaluate the performance of both models later.

We adopt two learning models to predict the response time for both local on-device and remote on-server executions in a dynamically changing environment where bandwidth, latency, data size, server's available CPU and memory change. Algorithm 6 presents the pseudo code for this dynamic application partitioning and training.

With Algorithm 6, we execute the application locally if there is no method to offload. If any method is offloaded, we monitor the response time while it is being executed on the server side. These values are used for training the learning models for the on-server response time. We also update the on-device learning model in a similar fashion when the methods (and application) are executed locally.

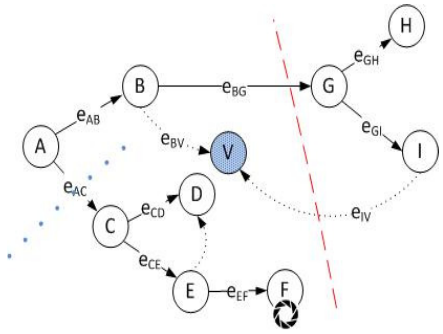


Fig. 3: Method Call Graph

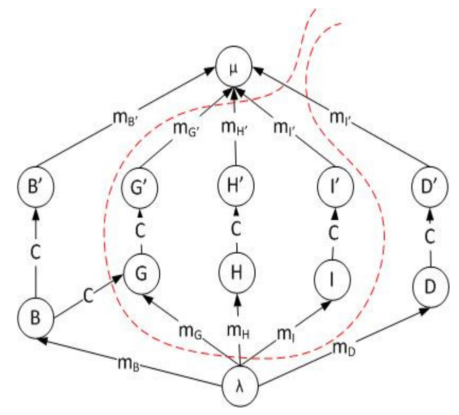


Fig. 4: Method Selection Graph

D. An Illustrative Example

In this subsection, we illustrate our algorithm with an example.

Figure 3 shows a method invocation diagram of an application. Here each white node represents a method and the solid arrow represents a method invocation incident from another method. The dark node represents a variable. The dotted arrow represents an access to a variable (global or class) modified by other methods. Each arrow has some associated cost with it. For example, when method A invokes another method B, A sends the parameters to that callee method B. When we offload the callee method, these parameters have to be sent to the server as well, which demands time to send it over the networks, which is denoted by e_{AB}. This cost can be found by monitoring the time when these parameters are sent to the server side from the mobile device. Similarly, if an offloaded method I accesses a global or class variable V modified by other method B, this variable has to be sent over the network so that the clone method on the server side can access these variables and execute flawlessly. We denote this cost as e_{VI}.

Suppose in this graph we find that method C and G consume most of the resources. If we offload C, methods D, E, and F are executing on the server side as well. Method F is accessing a camera, as a result it is not possible to execute method F to the server side. If we still decide to offload method C, the execution has to be transferred to the mobile device again while F is accessing the camera. To limit this back and forth executions, we should not offload C. To identify such constrained methods, our Algorithm 3 finds the list of classes, MCC of the reachable methods in RM of method C, which is $\{D, E, F\}$. As we have found that MCC of C includes a class accessing camera (from method F), we should not offload C.

Similarly, by offloading G, we are also executing H and I on the server side. So when method I is executed on the server side, it has to access variable V. As a result, while offloading method G, we have to send variable V to the server, which will cost us e_{IV}. In order to find such a set of variables, we deduce RV of method G. This set RV of method G has to be offloaded to the server side in order to offload the method G to the server. Note that here if G's VCC (the classes of RV) includes any of the constrained classes C, we do not offload G as well.

In addition, if we offload a method whose RV or RM is updated in parallel from other methods executing on the mobile device, we have to communicate again from the mobile device to the server. To minimize such overhead and inconsistency, we do not allow these methods to be offloaded. As illustrated in Figure 3, if any of method A, C, D, E, F or B is accessing the RM or RV of G, we do not offload G.

Thus, we find the `Method Call Graph` of B, G, H, and I and convert it to `Method Selection Graph` shown in Figure 4. In this graph, we introduce one additional node m' for each of the methods m. So for each of B, G, H, and I; we add B', G', H', and I' in Figure 4. We introduce two additional nodes λ and μ. The capacity of the edges between λ and B, G, H, I is set to their corresponding on-device execution time. The capacity of the edges between B', G', H', I' and μ is set to the corresponding on-server execution time. The capacity C between any other two nodes is set to the summation of the ranks of the methods in this graph ($C = (B_m + G_m + H_m + I_m + D_m) + (B_s + G_s + H_s + I_s + D_s)$). Figure 4 shows the corresponding `Method Selection Graph`. The dash curve line shows a min-cut partition where we are offloading the methods G, H, and I.

So far we have discussed our solution for optimizing the response time. The same approach can be utilized to optimize the energy consumption.

IV. Implementation

We implement a prototype of `Elicit` in Dalvik VM for Android applications. We have modified the Cyanogenmod [2] open source distribution to profile the applications without modifying the application to find the methods' members and the method call graph. After profiling, we partition the application optimally to achieve the highest gain regarding the system environment. Finally, we offload the method transparently to the server without modifying any source code or binary of the application.

A. Application Profiling

To profile applications, we have modified the instructions for a method invocation and method return. In Dalvik, whenever a method is invoked, it is translated to a method invocation instruction. The caller method's program counter and frame page are saved and then the callee method starts its execution. When the callee method finishes its execution, it returns to the caller method. The return instruction saves the return value to the caller method's return address and starts the caller method by retrieving the program counter and the frame pointer of the caller method. We have modified the method structure of the Dalvik VM so that it keeps records when a method is invoked and when it returns to the invoking method. From these two timestamps, we keep track of the execution time of the methods. We use PowerTutor [7] to measure the methods' energy consumption.

We build a tool based on `JAVA` to analyze the bytecode of the applications to construct the method call graph and thus find the list of variables and other methods accessed by the methods of an application. This one time analysis takes around

TABLE III: Description of Evaluated Applications

Application	Offloading candidate(s)	Total Number of methods	Description
DroidSlator [3]	→ `translate(String inWord, String toLang)` method of `ThaiDict` class : no global variable → `searchTELex(String word)` method of `DBAdapter` class : no global variable → `searchTE(final String word, String tableName)` method of `DBAdapter` class : db (GET) of `DBAdapter` class	100	Android translation application
MathDroid [5]	→ `computeAnswer(String query)` method of `Mathdroid` class : calculator (GET) of `Mathdroid` class → `evaluate(String stringExpression)` method of `Calculator` class : no global variable → `parse(String stringExpression)` method of `Calculator` class : no global variable → `simplify(Node expression)` method of `Calculator` class : no global variable → `evaluate(Calculator environment)` method of `Node`: no global variable	51	Android Calculator application
MatCal [4]	→ `times(Matrix B)` method of `Matrix` class : m (GET), n (GET), and A (GET) of `Matrix` class	24	Android application for matrix operation
NQueen	→ `findSolution(int board[],int n, int pos)` method of `NQueen` class : no global variable → `combination(int board[],int n, int pos)` method of `NQueen` class : no global variable → `check(int []board, int pos)` method of `NQueen` class : no global variable	10	Android application for NQueen problem
Picaso [6]	→ `project_and_compare(Bitmap bm)` method of `ReadMatlabData` class: no global variable → `project_and_compare(MMatrix testImage)` method of `ReadMatlabData` class: no global variable	22	Android face recognition application

30-40 seconds on an average for each of the applications. We deduce the byte code from the source code by `javap` command to disassemble the class files. Note we can also get the byte code from the applications [20] whenever the source code is not available.

To construct the method call graph, we analyze the byte code of each method and look for `invoke` or `invokevirtual` instructions to find the list of callee methods from a given caller method. In this way, we find the list of directly invoked methods and thus the list of reachable methods RM from Algorithm 1. From bytecode analysis, we also find the parent class of each method, and thus find MCC (Method Closure Class) from Algorithm 3.

Furthermore, whenever a method accesses a variable, it leverages two separate instructions (IGET and IPUT namely) to retrieve and save values. We also analyze the IGET and IPUT instructions to keep track of the variables (and their parent classes) that are accessed by the methods. To offload these methods, we have to send these parameters and variables (with IGET tag) to the server side. Once the method has successfully finished its execution and returns back to the mobile device, we save the return value and synchronize the modified variables (the variable having IPUT tag). Here if a method is accessing a variable related to mobile device's I/O, camera, or similar sensors, we do not offload it. We do this by examining the variables' parent class which is fetched from the byte code text.

In this way, we obtain the parameters for Algorithm 5. These parameters include the list of variables (and their classes) accessed by methods, the method call graph, methods' execution time, methods' parent classes, etc. To predict the on-device and on-server execution time by our learning model, we run the applications in the mobile device and offload them in different environments to the server for initial profiling. Then we conduct analysis to find the optimal partition of an application according to Algorithm 6. As discussed in section III-A, we have to discard the methods that access mobile device equipments (not exclusively, camera, sensor, view, etc.). To find this list, we populate a list from Android Java definitions. Based on the list of methods, their parent class, and the global and class variables (and parent class of these variables), we discard these methods (and their callers) from offloading.

We thus find the list of methods to offload along with the variables' states that must be synchronized before and after offloading. We intercept those methods' invocations and offload them accordingly as described in the next subsection.

B. *Offloading Mechanism and Decision Maker*

We keep the offloading mechanism transparent to the applications by adopting the transparent mechanism as proposed in POMAC [15]. Following the same principles, we trap the method invocation instruction and gets the parameters and variables required by the method to execute on the server side. POMAC [15] can offload the methods which do not access any class or global variable. POMAC only retrieves the methods' input parameters and sends them to the server for remote ecxecution. Moreover, POMAC requires the methods to be identified beforehand for offloading. In our work, `Elicit` finds the resource-intensive method automatically and makes sure that these methods are un-constrained by byte code analysis. In addition to that, `Elicit` can offload methods which access the class or global variables in addition to the method input parameters.

We pack these parameters and ship them to the server side for execution. On the server side, we deploy Android

ASUS R2 virtual machine in VirtualBox [8]. This server-side android VM has all the applications' apk files which are uploaded in prior. We use Java reflection to dynamically load the appropriate applications and its classes to execute the offloaded methods on the server side virtual machine when needed. Once the server completes the execution, we return the result (and the states and variables that must be synchronized with the mobile client) to the mobile device. Once the mobile device gets the results back from the server, it synchronizes itself and the offloaded method returns to the caller method and the application follows its original flow of execution. For the decision maker, POMAC [15] suggested that Multilayer Perceptron has better performance, so we have also implemented the Multilayer Perceptron in our prototype. In addition to that, we compare MLP with other classifiers in the evaluation. We have implemented the learners in Weka [9].

V. Performance Evaluation

We evaluate our `Elicit` prototype with five different Android applications. Table III describes the applications and their functionality. We choose these applications considering their different characteristics: DroidSLator, MatCal, and Picaso are both computation- and data-intensive. NQueen are purely computation-intensive while MathDroid is data-intensive. Moreover, most of them are the applications being evaluated in the previous studies, where the offloaded methods were hand-picked. By evaluating them in `Elicit`, we are able to tell whether `Elicit` can efficiently identify the same method for offloading.

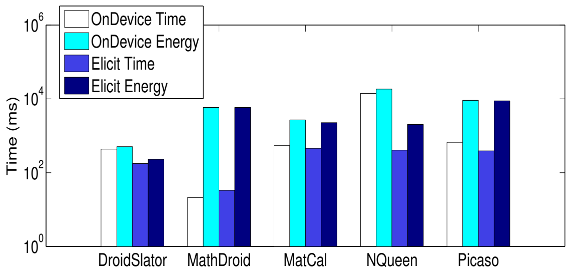


Fig. 6: Response Time And Energy Consumption of Different Applications

For each of these applications, we first find the optimal partition of the applications and find the method to be offloaded. Table III shows the methods which are found to be optimal to be offloaded based on Algorithm 6. In the table, each partition method shows the method name, parameter list, and global variables that are accessed by this method (and its subsequent methods). Each global variable has a tag IGET or IPUT, which indicates whether a variable is accessed by the method (IGET) or modified by the method (IPUT).

In our experiments, `Elicit` indeed finds similar methods as the candidate to be offloaded as in previous studies. For example, in Young et al. [18], the DrdoidSlator application [3] was evaluated. Our candidate is the `translate` (and its subsequent methods: `searchTELex` and `searchTE`) method, which was found manually to be the most resource intensive method in [18]. Similarly, for MatCal [4] and MathDroid [5], we find that the applications are partitioned in the `times` and `computeAnswer` (and its subsequent) methods which were also found to be the most resource intensive methods in POMAC [15]. For Picaso, we found similar candidate methods as mentioned in [15]. Here we have added one new application, NQueen, for which we have found `findSolution` (and its subsequent) methods to be offloaded according to the optimal cut. Note that here each application has many methods (even a hundred excluding the basic Java methods like `String Compare` or `println` as shown in Table III), and we are offloading the most resource consuming method(s) of the application for optimal performance.

Next, we experiment `Elicit` in different environments. We want to (1) empirically evaluate the offloading performance and (2) evaluate that the methods suggested by `Elicit` for offloading can improve the response time. We evaluate our prototype in a Google Nexus one with 1 GHz CPU and 512 MB memory. We have five different configurations in the experiments. In each configuration, the CPU, memory, data size along with the network parameters are dynamically changing. To simulate the Fog computing, we have LAN, WLAN, and 802.11 settings which represents Fog settings, while the others (4G and 3G) simulate the cloud setting. In the *LAN* setting, we keep the CPU availability at 2 GHz and memory at 1 GB in the server. We have also set the bandwidth and the latency between the smartphone and the server to 100 Mpbs and 20 ms, respectively. In the *WLAN* setting, the CPU availability and memory in the server are set at 1 GHz and 2 GB, respectively, where the bandwidth between the smartphone and the server is 30 Mbps and the latency is 20 ms. In the *802.11g* setting, the bandwidth is kept at 25 Mbps while the latency is 50 ms. The CPU availability is 2 GHz and the memory is 2 GB in the server. In the *4G* environment, the bandwidth and latency is set to 5 Mbps and 75 ms where for *3G* these values are 500 Kbps and 200 ms. The CPU availability is set at 2 GHz and the memory is set at 2 GB in the server for both *3G* and *4G*. Our goal here is to check how our algorithm works in dynamically changing environments, so we change the server side CPU and memory as well. We emulate different network configurations by changing the network bandwidth and latency with `traffic control` utility while using the wifi of the Google Nexus One. In this way, we emulate 4G and 3G in the Google Nexus One.

We have conducted ten experiments in each of the environmental setup. Table IV shows the gain ratios of the applications in different environments and Figure 5 shows the different data size for different applications. 10 experiments are conducted in each of the five configurations as mentioned before. So our experiments are conducted when the data size, network bandwidth and latency, and server-side CPU and memory availability change for each of the instances (whenever possible) for each application. Note that the training and classification overhead of the MLP has been accounted. The final average of the gain ratios is shown in the last row. We find that for DroidSlator, Picaso, MatCal, MathDroid, and NQueen; the gain ratios are greater than 1. MathDroid has both ratios very close to one. For MathDroid, it does not save that much time and energy by offloading. For all applications, mostly the gain ratios are higher in Fog settings (LAN and WLAN) compared to the cloud (3G or 4G), which is expected, as the bandwidth gets higher and the latency gets lower, our Algorithm 6 can find a better partition to

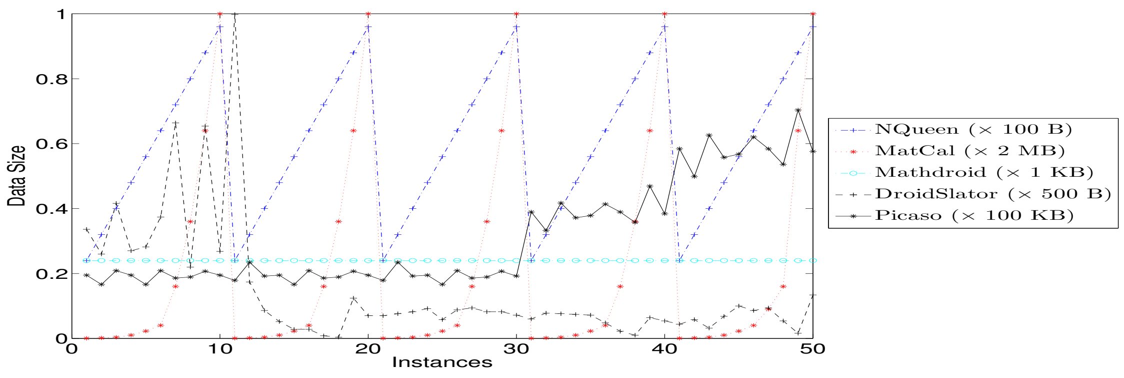


Fig. 5: DataSize for all the 50 experiments for different applications

TABLE IV: Gain Ratios of the Applications in Different Environments

	DroidSlator		MathDroid		MatCal		NQueen		Picaso	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
LAN	5.30	4.3	1.05	0.97	1.45	1.50	19.62	11.10	4.14	1.06
WLAN	3.59	2.8	0.99	0.96	1.37	1.40	19.61	11.01	3.45	1.07
802.11	0.98	1.1	1.09	1.07	1.23	1.18	19.44	10.99	3.32	1.07
4G	3.72	2.7	0.99	0.97	1.13	1.01	19.26	10.89	1.01	0.98
3G	0.97	0.98	1.05	1.03	0.98	0.97	18.93	10.71	1.02	0.97
Average	2.91	2.38	1.02	1.01	1.23	1.21	19.37	10.94	2.58	1.03

save energy and response time and offload them accordingly. The better network condition also enhances the performance of the offloaded methods. Here we have found that network condition plays a very important role on the offloaded methods' performance, which is consistent with previous studies [13] and [15].

Figure 6 shows the average response time and energy consumption of the 50 experiments in the 5 different environments. In this figure, the y-axis shows the response time in millisecond and energy consumption in millijoule. Note that y-axis is in log scale. Figure 6 shows that by offloading: DroidSlator and NQueen can save significant time and energy. `Elicit` reduces 2.91x and 19.37x response time of DroidSlator and NQueen compared to the on-device execution. `Elicit` can also save 2.38x and 10.94x energy consumption for these applications, respectively. We also found that `Elicit` can not save significant time or energy for the MathDroid application. In fact we have found that most of the time it is optimal to execute MathDroid on the device itself. For Picaso, `Elicit` saves a lot of time (2.58x) but slightly increases the energy savings (1.03x). For MatCal, `Elicit` can save a moderate amount of time and energy, the gain ratio is 1.23x for time and 1.21x for energy, respectively.

So far, the results are based on MLP. The SVM-based evaluation shows similar results. We omit for brevity.

A. *Comparing with Other Learning Models*

Several classifiers have been used in previous studies, such as Threshold [18] and Linear-Regression [13]. In the evaluation, we also compare MLP's performance against those, including Linear Regression, Support Vector Machine (SVM), and Decision Tree. We could not compare with Threshold Based policy because it does not support value prediction. Similar to MLP, we train each of the learners with 50% of all the instances of these applications (DroidSlator, MathDroid, MatCal, NQueen, and Picaso) and test with the rest of the dataset and make the decision accordingly in the mobile device. Table V shows the average response time and energy consumption of different models for these applications for 50 instances of each application. Table V shows that most of the time MLP has better performance than the other learners. Again SVM achieves similar results.

Although for some scenarios LR is performing better than MLP or SVM, overall MLP and SVM have better performance than LR. Moreover, in Table V, we can see that for MathDroid, all of the models are performing almost the same. This is because the optimal performance is achieved when MathDroid is executing on the device locally, which all the models predict to do. The difference mainly comes from different classification and training overhead of different classifiers.

VI. Related Work

Previous research [12], [22], [13], [21], [10], [14] has investigated how to partition mobile applications and offload computing-intensive tasks to the more powerful counterparts such as clouds and servers. Balan et al. [10] focused on how easily applications can be modified for offloading. Odessa [21] showed that offloading can improve the response time for streaming and pipelined applications by three times. Thinkair [17] provides API for computation offloading which requires special compilation. In addition to that, the computation-intensive methods are pre-indicated and the developer has to make sure that they are appropriate for offloading beforehand. Some other research [13], [18] focused

TABLE V: Average Response Time (ms) and Energy consumption (mJ)

	DroidSlator		MathDroid		MatCal		NQueen		Picaso	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
MLP	176.58	230.73	32.93	5850.33	458.16	2262.96	409.35	2033	388.08	8793.33
Linear	190.72	263.40	33.32	5846.67	519.72	2262.67	420.90	1963	435.47	9152.67
SVM	189.42	263.75	31.02	5847.00	463.40	2225.00	449.7	1883	433.18	9153.00
DT	190.03	262.40	32.63	5845.67	465.13	2265.67	451.9	1886	387.77	8793.67

on application level partitioning. While making the offloading decision, MAUI [13] adopted a linear model to make the offloading decision, while Odessa [21] leveraged previous execution statistics and used the index of the ratio-of-benefit to make the offloading decision. Young et al. [18] proposed a static threshold to make offloading decisions. When the amount of data to be transferred is greater than a certain value, the execution is offloaded to the more powerful counterpart. Comet [14] also proposed a threshold based policy for offloading computation: when the thread execution time exceeds twice of the RTT to the server, then it migrates the thread. Zhang et al. [24] formulated the problem of partitioning application as the shortest path problem, which does not consider constraints of real-world applications. Zhang et. al. [25] considers the problem in the class level and make the offloading decision based on LOC.

Our work differs from existing ones in that first we consider the problem that was not well studied before, i.e., how to find the appropriate methods of a mobile application to offload. For each application, potentially there are hundreds of methods that can be offloaded. In addition to that, some resource constrained methods can not be offloaded as they may access the camera or sensors of the mobile devices. Furthermore, our proposed work can dynamically select the most appropriate methods for offloading based on available resources at runtime.

VII. Conclusion

The increasing popularity of smartphones are driving the fast development of mobile applications. For resource-intensive mobile applications, there is an increasing demand to offload to more powerful counterpart, such as clouds and nearby servers via fog computing. While most of the existing studies have focused on how to offload, little research has been conducted on how to automatically find the most appropriate methods to offload. In this study, we have designed and implemented `Elicit`, a framework to efficiently partition the applications in an optimal way to find the appropriate methods for offloading dynamically based on runtime resource availability. Extensive experiments have been conducted to evaluate `Elicit` and the results show that `Elicit` can work with existing real-world mobile applications and efficiently find the best offloading methods to reduce the response time and energy consumption.

References

[1] Amazon Silk. http://amazonsilk.wordpress.com/.

[2] CyanogenMod. http://wiki.cyanogenmod.org/w/Build_for_passion.

[3] Droidslator. http://code.google.com/p/droidslator/.

[4] MatCal. https://github.com/kc1212/matcalc.

[5] MathDroid. https://play.google.com/store/apps/details?id=org.jessies.mathdroid&hl=en.

[6] Picaso. http://code.google.com/p/picaso-eigenfaces/.

[7] Power Tutor. www.powertutor.org/.

[8] Virtual Box. https://www.virtualbox.org/.

[9] Weka. http://www.cs.waikato.ac.nz/ml/weka/.

[10] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *Proc. of Mobisys*, San Juan, Puerto Rico, June 2007.

[11] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[12] B.G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proc. of EuroSys*, pages 301–314, 2011.

[13] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proc. of MobiSys*, San Francisco, CA, USA, June 2010.

[14] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: code offload by migrating execution transparently. In *OSDI*, 2012.

[15] Mohammed Anowarul Hassan, Kshitiz Bhattarai, Qi Wei, and Songqing Chen. Pomac: Properly offloading mobile applications to clouds. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, Philadelphia, PA, June 2014.

[16] Jon Kleinberg and Éva Tardos. *Algorithm design*. Pearson Education India, 2006.

[17] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[18] Y. W. Kwon and E. Tilevich. Power-efficient and fault-tolerant distributed mobile execution. In *Proc. of ICDCS*, 2012.

[19] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. Incremental support vector learning: Analysis, implementation and applications. *The Journal of Machine Learning Research*, 7:1909–1936, 2006.

[20] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[21] M.R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. of Mobisys*, pages 43–56. ACM, 2011.

[22] M. Satyanarayanan, P. Bahl, R. Caceres, and N.l Davies. The case for VM-based cloudlets in mobile computing. In *IEEE Pervasive Computing*, volume 8(4), October 2009.

[23] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[24] Weiwen Zhang, Yonggang Wen, and Dapeng Oliver Wu. Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. In *INFOCOM, 2013 Proceedings IEEE*, pages 190–194. IEEE, 2013.

[25] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *ACM SIGPLAN Notices*, volume 47, pages 233–248. ACM, 2012.