POMAC: Properly Offloading Mobile Applications to Clouds

Mohammed A. Hassan[1], Kshitiz Bhattarai[2], Qi Wei[1] and Songqing Chen[1]

[1]{mhassanb,qwei2,sqchen}@gmu.edu, Dept. of Computer Science, George Mason University
[2]kshitiz.bhattarai@sap.com, SAP Lab, Palo Alto, CA

Abstract

Prior research on mobile computation offloading has mainly focused on *how* to offload as well as *what* to offload. However, the problem of *whether* the offloading should be done attracted much less attention. In addition, existing offloading schemes either require special compilation or modification to the applications' source code or binary, making them difficult to be deployed in practice. In this work, we introduce POMAC, a framework to enable *dynamic* and *transparent* mobile application offloading to clouds. A prototype has been implemented on the Dalvik virtual machine and our preliminary evaluations show that POMAC can outperform existing schemes significantly and work with real-world applications seamlessly.

1 Introduction

Mobile devices and mobile applications are increasingly popular today. Due to constrained on-device resources, a lot of efforts have been made to offload mobile applications partially or fully to more power servers, such as clouds. For example, prior research [10, 17, 11, 16, 15] has proposed to partition the application and offload computation-intensive portions to more powerful counterparts such as servers [11, 15] or cloned VMs [10, 17], while application-specific offloading is considered in [16]. To implement the offloading, these models often require the applications or the binary executables to be modified [11, 15], or require special compilation [14]. Clone-based approaches [10] can offload applications without modifications to applications, but they require a full mobile image running on the cloud.

However, existing research has paid little attention to whether the offloading should be done. For example, Young et al. [15] suggested to offload the computation when the data size to transfer is greater than 6 MB, while MAUI [11] makes offloading decisions based on a linear regression model between selected features. But in practice we find that these simple approaches result in more than 50% wrong decisions, which leads to more energy consumption and longer response time. In addition, existing schemes either require special compilation [14] or modifications to the applications' source code or binary [11, 15], making them difficult to be adapted in practice.

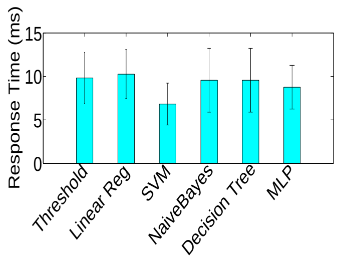
In this paper, we aim to build a framework, POMAC: **P**roperly **O**ffloading **M**obile **A**pplications to **C**louds, to address both of the above issues. For the former, we empirically identify pivotal features that play key roles in making the offloading decision and compare the performance of different decision makers. For the latter, we design a transparent offloading mechanism through method interception at Dalvik virtual machine (VM) level to allow mobile applications to offload their computation intensive methods without requiring any special compilation or any modifications to the application's source code or binary executable.

A prototype is implemented on Android Dalvik virtual machine. We evaluate it with a few popular Android applications from Google Play. The results show that `POMAC` can work seamlessly with real-world applications and outperform the existing schemes significantly.

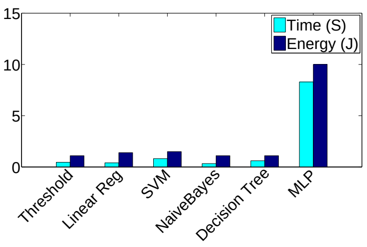
The rest of the paper is organized as follows. After presenting offloading challenges in section 2, we discuss POMAC design and implementation in section 3. Some preliminary results are discussed in 4 and we make concluding remarks in section 5.

2 Offloading Issues

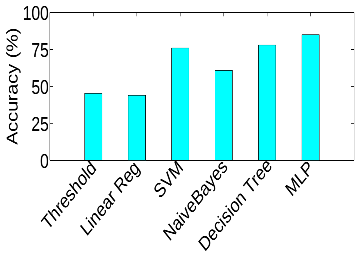
The main challenges associated with an offloading model include whether the offloading is worthwhile, and how to do offloading if it is worthy. A decision maker should determine the first part, while an proper offloading mechanism should be responsible for the second part. However, existing research addressing these challenges has some the following limitations.



(a) Classification Time



(b) Time and Energy Consumption



(c) Accuracy

Figure 1: Comparison betweeen different classifiers

2.1 Problems in Decision Making

Clearly, an offloading is worthwhile only when the local (on-device) execution consumes more time and energy than the offloading overhead. Many factors can impact the above aspects [11, 13, 17] dynamically. Thus, a simple policy like always-offloading may deteriorate the application's performance.

For example, some prior studies (e.g., [15]) suggested a threshold based policy for making the offloading decision. A method is offloaded only when its parameter data size is greater than a pre-defined threshold value. Apparently, it is difficult to set a generic threshold as it is often application-dependent. More importantly, this might work in a static environment where the resource information is stable, such as stable bandwidth between the sever and the mobile device. For a mobile connection where conditions change continuously and dynamically, it is hard to make the offloading decision solely based on a static threshold. Moreover, a threshold based policy always makes the same offloading decision without considering network bandwidth or latency, let alone other factors such as memory or CPU availability.

A linear regression model [11] was also used to capture the relationship between the system features. A linear regression model works well when only a limited number of features are changing, such as the bandwidth or latency as considered in MAUI [11]. But the complexities among the interplay of more features cannot be captured by a linear regression model. Our experimental results with Android Google Nexus one (with 1 GHz CPU and 512 M memory) and a virtual machine (with 2 GHz CPU and 2 GB memory) on the server and DroidSlator [1] show that a linear regression model has a root relative absolute error rate of 58.01%, while it is only 17.66% for another classifier named support vector machine (SVM). It also shows that a threshold based classifier can lead to 50% wrong decisions (we omit their details for brevity).

Therefore, to correctly make the offloading decision, we need a classifier that should have the properties to be highly variant, low biased, and tolerant of noise. That is, a good classifier should be i) lightweight to run on mobile devices, ii) able to capture the interplay between key features, iii) highly accurate and iv) self-learning gradually over time.

2.2 Problems in Offloading Mechanisms

Many offloading mechanisms have been proposed to offload the computation intensive part of mobile applications to a more powerful counterpart. In general, these schemes can be classified in two broad categories: i) offloading by application partitioning and ii) offloading by full VM cloning.

Prior research [11, 8, 9, 16, 15] achieving offloading by application partitioning either switches the execution [8, 16] or invokes RPC [11, 15] when approaching any resource intensive segment in the code. While these approaches can save significant time and energy for mobile applications, they require either annotation [11], binary modification [15], or special compilation [14], which requires modifications to every application so as to fit into the offloading framework.

VM based cloning maintains a full clone of the smartphone image in the cloud [10, 12, 17]. To offload computation intensive segment of the applications, these approaches suspend the smartphone execution and transfer the execution in the VM clone in the Cloudlets. While transferring, it requires a lot of data transfer ($\sim$100 MB) for synchronization.

Thus, to minimize the offloading overhead and to support the large number of existing mobile applications that do not have offloading logic, we argue that a transparent offloading approach at the method level sometimes may be more desirable. As a transparent scheme, it should not require any modifications to the applications source code or binary executables or special compilation.

3 POMAC Design and Implementation

In this section, we describe the design and implementation of our offloading model POMAC.

3.1 Features and Classifiers selection

To find effectiveness of different classifiers for making offloading decision, we compare the accuracy of different classifiers. Figure 1 shows the result of different classifiers we have examined with our data set of Picaso [4] collected from three different users. We implement the classifiers using open source library Weka [6] on the Android phone Google Nexus one with 512 MB memory and 1 GHz CPU. We have trained the classifiers with 50% of the data (for Picaso [4]) because more training could help some classifiers. Figure 1(a) shows that the time to classify one instance remains almost the same for all the classifiers, but multi-layer perceptron (MLP) takes more time and energy to train (Figure 1(b)). Obviously higher dimensional classifiers like MLP, SVM, and Decision Tree (DT) achieve better accuracy (around 80%) than the other lower dimensional classifiers (Figure 1(c)). But the inherent limitation of Decision Tree and SVM is that, these two classifiers don't support online training. Although MLP consumes more time and energy for training, this cost is amortized over time and becomes less significant. So in our current prototype we are using MLP as the classifier.

To make the offloading decision, it is important to find out the pivotal features those impact the response time and energy consumption of an offloaded method. In addition to bandwidth, latency, and data size [11, 15], we have considered the CPU and memory availability both on the server side and the mobile device while making offloading decision.

3.2 Offloading Mechanisms

Once an offloading decision is made, the next step is to perform the offloading of the computation intensive part transparently.

To offload a method of an application, we need to retrieve the method parameters and send them to the server side for remote execution. Since Android applications run on Dalvik VM, we propose to intercept the method invocation instruction of the application VM (Dalvik VM) and redirect the execution towards the server side. Here the main challenge boils down to intercepting the method invocation instruction and retrieving the parameters. Once the method is executed remotely with the provided parameters, the result is sent back to the caller of the method to continue its execution on the mobile device.

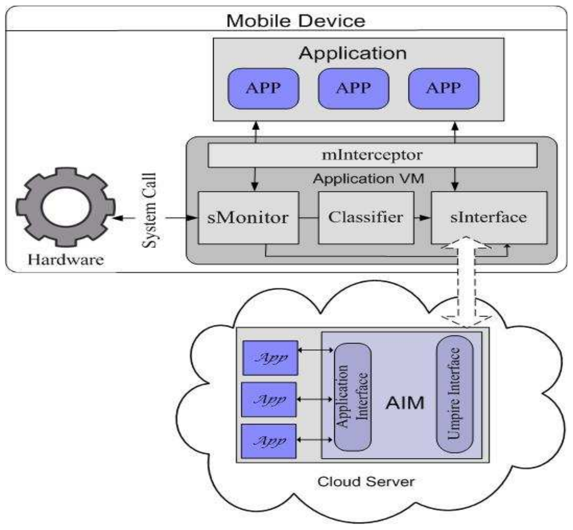


Figure 2: Components of POMAC

Thus, POMAC intercepts the method invocation at the Dalvik VM instruction level. Figure 2 shows different components of POMAC; including mInterceptor, sMonitor, Classifier, and sInterface. While intercepting, POMAC gets the current executing methods' class and parameter list, return type, etc through mInterceptor. If a decision is made to offload the method, it serializes the parameters and sends it to the sInterface to offload the method along with the parameters. After the server side finishes the execution, POMAC gets the result back from sInterface, deserializes the result to build the appropriate object and returns to the invoker of the method. In case of a local execution or a remote server failure, POMAC lets the application continue its normal execution flow.

In Dalvik, each kernel thread is dispatched to one user level thread and the thread proceeds instruction by instruction. Similar to any other instruction, a method invocation has its own instruction (Dalvik opcode). As aforementioned, POMAC intercepts the methods at Dalvik VM instruction level. The main challenge here is to intercept the Dalvik interpretor at the point where it invokes a method. We find that when Dalvik interpretor reaches any method invoking instruction, it jumps to the address of the method's instruction and starts to execute the method. Before jumping to the method's instruction, Dalvik saves the frame page and the program counter of the current method to restart from the same point after returning from the invoked method. The input parameters of the invoked method are saved in the invoked method's frame page. The frame page saves the values for primitive and the address of the object input parameters. Once the input parameters are set, Dalvik jumps to the invoked method's instruction to start execution. At this point, `mInterceptor` intercepts the method call.

If an offloading decision is made, `POMAC` gets all the fields' value of the input parameters and packs them into a byte array. The name of the application, method, and class; the parameters' type are also encoded in the byte array. Once packed, `mInterceptor` sends the byte stream to the `sInterface` and suspends for the return value. The `sInterface` communicates with the server side to execute with the provided parameters and gets the result back. After getting the return object, `mInterceptor` returns it to the invoking method. To return from the invoked method, `mInterceptor` puts the value (for primitive data types) or the address (for other objects) in the `return value` field of the Dalvik interpreter. Similar to the `program counter` of `frame page`, the `return value` is a field of the current executing thread where the returned entity from an invoked method is saved. `mInterceptor` then re-starts normal execution of the application. It first retrieves the invoking method's frame page and the program counter (which were saved before), and starts the execution from where it was suspended.

`POMAC` also dynamically collects the system environment and resource information, such as network bandwidth, latency, and CPU availability at real-time via `sMonitor` component. Currently we send two small packets to the server back to back to measure the latency and bandwidth. To get the CPU and memory availability, we examine the */proc/stat* and */proc/meminfo* files in Android and server (both run on Unix OS). The data size can be calculated from the input parameters of the invoked method. Such information is the input to the `Classifier` for making the offloading decision dynamically.

The `Classifier` is the brain of `POMAC`. It uses MLP to accurately characterize the relationships among different factors impacting the offloading performance. Whenever a potential candidate method is intercepted, the `Classifier` takes the method parameters, calculates the size of the arguments, and considers the feature values collected by `sMonitor`. Based on collected information, it makes the decision either to offload the method or to execute it locally. The classifier has a feedback channel as well so that it can learn by itself through the entire decision process.

On the server side, `POMAC` requires the same application to be installed (not running) so that the server can invoke and execute the same method that is offloaded. The mobile applications can be installed and executed on the application VM or on the desktop machines if supported. The server-side receives the offloading requests along with the parameters through the `POMAC Interface`, and instantiates the appropriate method class at runtime, creates the input parameters from the provided information from the mobile device, and executes the method through `Application Interface`. Once the execution is completed, the server returns the result back to the mobile device through the `POMAC Interface`.

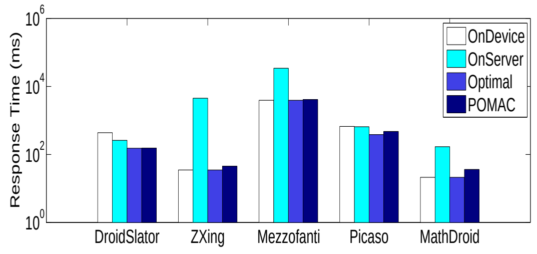
4 Preliminary Evaluation

In this section, we present some preliminary performance results of `POMAC` in a dynamically changing environment where the network (bandwidth and latency) and server capacity (available CPU and memory) changes.

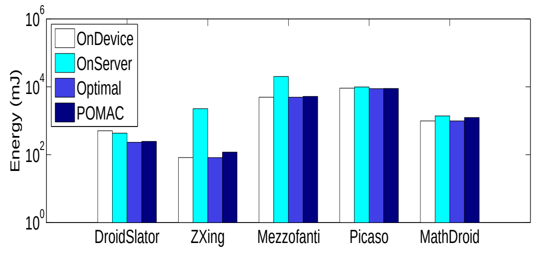
We evaluate `POMAC` with five popular Android applications. Currently, we manually identify the most resource intensive methods of these application and offload them when necessary. The applications and the corresponding method candidates are as follows. In DroidSlator [1] (Android translation application), the candidate method is `translate(String inWord, String toLang)` of the `ThaiDict` class, while for Zebra Xing(ZXing) [7] (Android product bar code reader), it is the `decodeWithState (BinaryBitmap)` method of the `MultiFormatReader` class. Mezzofanti [3] (Android optical character recognition application) has most of its computation in the `ImgOCRAndFilter(int[] bw_img, int width, int height, boolean bHorizontalDisp, boolean bLineMode)` method of the `OCR` class. In Picaso [4] (Android face recognition application), the candidate method is `project_and_compare(Bitmap bm)` of the `ReadMatlabData` class; while for MathDroid [2] (Android Calculator application), it is the `computeAnswer(String query)` method of the `MathDroid` class.

We have five different configurations in the experiments. In each configuration, the CPU, memory, data size along with the network parameters are dynamically changing. In *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 *WLAN* setting, the CPU availability and memory in the server is 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 *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 *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*.

Figure 3 shows the average response time and energy consumption of the the applications when different decision-making approaches are adapted for 50 instances of each application. 10 experiments are conducted in each setting. We use `PowerTutor` [5] to measure the power consumption of the applications.



(a) Response Time



(b) Energy Consumption

Figure 3: Application Offloading Performance

Figure 3 shows our preliminary results. Compared to always-offloading (OnServer), the user's response time with `POMAC` is reduced to 59.39%, 1.00%, 11.99%, 72.28%, and 21.44% of the response time if DroidSlator, ZXing, Mezzofanti, Picaso, and MathDroid applications are always offloaded to the server. Correspondingly, in `POMAC`, the energy consumed on the smartphone is 57.21%, 9.49%, 25.80%, 90.01%, and 77.97% of the energy consumed if these applications are always offloaded.

On the other hand, compared to OnDevice where the applications are always executed locally, `POMAC` also provides the user faster response time for DroidSlator and Picaso (35.40% and 71.25% of their local execution time, respectively). The corresponding energy consumption is about 48.61% and 97.48%, respectively, of the energy consumption when they are executed locally. For Mezzofanti, `POMAC's` performance is similar to that of the local execution.

Compared to the Optimal policy, `POMAC` is expected to perform worse because of the training and classification overhead along with some misclassifications. However, for most of the applications, `POMAC` achieves the closest results to those of the Optimal policy. As shown in the figure, `POMAC` takes 1.46%, 29.54%, 4.82%, 23.47%, and 69.56% more time than the Optimal policy for DroidSlator, ZXing, Mezzofanti, Picaso, and MathDroid; and it consumes 6.03%, 45.12%, 5.23%, 1.02%, and 9.67% more energy of that consumed with the Optimal policy.

Among the evaluated applications, `POMAC` does not perform well for ZXing and MathDroid. This is because ZXing is purely data intensive and always has the best performance when executing on device [15]. MathDroid is neither data- nor computation-intensive. Looking into the applications, we find execution time on the device is very small. Thus training and classification time in `POMAC` becomes significant overhead compared to its actual execution time (in terms of percentage). If we look at the absolute values, they are very small (e.g., 119mJ for ZXing under `POMAC`).

Figure 3 also re-confirms that a static policy of always offloading or the default on-device executions may work for some applications, but not for other applications. In contrast, following the optimal policy can always lead to the best performance, while `POMAC` offers near-optimal performance.

5 Conclusion and Future Work

In this paper, we set to investigate two issues for offloading mobile applications to clouds. We first consider whether an offloading should be conducted and then discuss how to transparently offload the task if offloading is worthwhile. Compared to the existing decision-makers utilizing a static policy, we have shown that a static policy may worse the application's performance. Thus, we have designed and implemented a decision maker based on MLP that can capture the dynamic nature of the relevant key factors and their relationship. Furthermore, we have implemented a transparent offloading mechanism at the method level in our system `POMAC`. Compared to existing schemes, `POMAC` does not require any modifications to the source code or binary, or special compilation. Preliminary experiments have been conducted to evaluate `POMAC`. The results show that `POMAC` can work near-optimally with existing mobile applications.

To offload a method, `POMAC` demands network communication from the Dalvik VM, which requires the application to allow network communications for permission checking, which is a violation of the application policy of network usages. Also, currently, `POMAC` works with applications where the offloaded method does not refer to any global or class variables. This makes our implementation simple. We are going to address these issues in our next step. We also plan to investigate more to find the more appropriate classifier in our future work.

6 Acknowledgement

We appreciate constructive comments from anonymous referees. The work is partially supported by NSF under grants CNS-0746649 and CNS-1117300.

References

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

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

[3] Mezzofanti. `http://code.google.com/p/mezzofanti/`.

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

[5] Power Tutor. `www.powertutor.org/`.

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

[7] ZXing. `http://code.google.com/p/zxing/`.

[8] 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.

[9] Rajesh Balan, Jason Flinn, M. Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. The case of cyber foraging. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, July 2002.

[10] 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.

[11] 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.

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

[13] M. A. Hassan and S. Chen. An investigation of different computing sources for mobile application outsourcing on the road. In *Proc. of Mobilware*, June 2011.

[14] 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.

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

[16] 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.

[17] 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.