Light-Weight Adaptive Task Offloading from Smartphones to Nearby Computational Resources

Shigeru Imai
Rensselaer Polytechnic Institute
110 Eighth Street
Troy, NY 12180
+1 (518) 428-8870
imais@rpi.edu

Carlos A. Varela
Rensselaer Polytechnic Institute
110 Eighth Street
Troy, NY 12180
+1 (518) 276-6912
cvarela@cs.rpi.edu

ABSTRACT
Applications on smartphones are extremely popular as users can download and install them very easily from a service provider's application repository. Most of the applications are thoroughly tested and verified on a target smartphone platform; however, some applications could be very computationally intensive and overload the smartphone's resource capability. In this paper, we describe a method to predict the total processing time when offloading part of an application from smartphones to nearby servers. In our method, if an application developer can (1) define a basic model of the problem (*e.g., f(x)=ax+b*) and (2) implement an algorithm to update the model (*e.g.,* least squares method), the application quickly adjusts the parameters of the model and minimizes the difference between predicted and measured performance adaptively. This accurate prediction helps dynamically determine whether or not it is worth offloading tasks and the expected performance improvement. Since this model's simplicity greatly reduces the time required for profiling the performance of the application at run-time, it enables users to start using an application without pre-computing a performance profile. Our experiments show that our update parameter protocol for the performance prediction functions works sufficiently well for a face detection problem. The protocol requires on average 7.8 trials to update prediction parameters, and the prediction error stays less than 10% for the rest of the trials. By offloading the face detection task to a nearby server for an image of 1.2Mbytes, the performance improved from 19 seconds to 4 seconds. This research opens up the possibility of new applications in real-time smartphone data processing by harnessing nearby computational resources.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks]:** Distributed Systems—*Client/Server*

General Terms
Algorithms, Performance, Design, Experimentation

Keywords
Mobile computing, smartphone, task offloading

RACS 2011, November 2-5, 2011, Miami, FL, USA.

1. INTRODUCTION
In recent years, applications running on smartphones, such as Apple iPhone and Google Android Phones, have become very popular. Most of the released applications work fine on those platforms, but some of the applications including video encoding/decoding, image recognition, and 3D graphics rendering, could take a significant amount of time due to their computationally intensive nature. Processors on mobile devices are gradually getting faster year by year; however, without aid from special purpose hardware, they may not be fast enough for those computationally intensive applications.

To improve the user experience of such computationally intensive applications, offloading tasks to nearby servers is a common approach in mobile computing. Consequently, a lot of research has been done including [1], which introduces an offloading scheme consisting of a compiler-based tool that partitions an ordinary program into a client-server distributed program. More recent efforts utilize cloud computing technology for task offloading from mobile devices [2][3]. Both partition an application and offload part of the application execution from mobile devices to device clones in a cloud by VM migration.

On one hand, these previous studies are very flexible in terms of partitioning the program and finding the optimal execution of distributed programs; also, such methods could be applicable to a wide range of software applications. On the other hand, for simple single-purpose applications, which are typical in smartphone applications, these techniques may be too complex as partitioning, profiling, optimizing, and migrating the distributed program significantly increase running time. If an application is simple enough and a developer of the application is aware of the blocks that take most of the time, it would be reasonable to statically partition the program. Also, if application developers have done experiments on a target problem and are knowledgeable about the characteristics of the problem, they could formulate a function to predict the performance or cost of the application execution with less effort in run-time profiling.

In this paper, we describe a simple model to predict the performance of distributed programs. The model updates prediction parameters on each run to adapt to network and server speed changes in a very light-weight manner. In particular, the model does not require analyzing and profiling the application developer's original program, but it assumes the program contains a basic model that reflects the nature of the task to predict the performance. We present the design of the model as well as implementation and evaluation with a face detection problem running on an iPhone as an example. Our evaluation shows that

the model predicts the actual performance very well as we iterate problem executions.

This paper continues as follows. Section 2 introduces the motivation of this work with a preliminary result of face detection performance on an iPhone as a smartphone platform. Section 3 describes our model that predicts the performance of distributed programs. Section 4 illustrates an instance of the model when applying it to face detection. Section 5 investigates the accuracy of the prediction generated from the model instantiated in Section 4. Section 6 and 7 discusses related work and possible future work respectively. Finally, Section 8 concludes our work.

2. MOTIVATION

2.1 Sample Application: Face Detection

Face detection is an application of image processing that determines the locations and sizes of human faces on given digital images as shown in Figure 1. It is commonly used in digital cameras that detect faces to make those faces look better through image processing. Another example is a photo sharing service; such as Google Picasa or Facebook, which helps users add name tags for detected faces on photos. In those uses of face detection, the processing speed is pretty fast; users may even do not notice when those photos are processed because they are processed on fast hardware for image processing or on servers where plenty of computational resources are available.

However, there may be a case that users want to detect faces right after they take a photo using smartphones. One example of such usage is a make-up game application where the user can try various make-ups or hair styles with the photo without specifying a region of the face. For this kind of application, immediate face detection is critical for a good user experience.

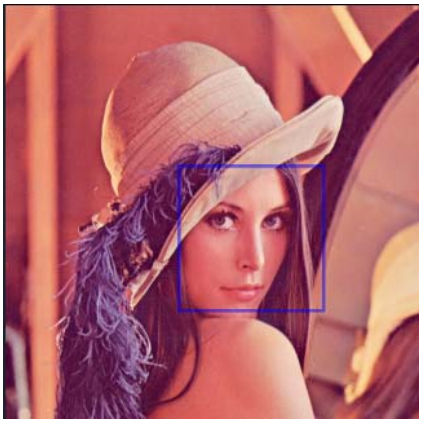

Figure 1. Example Result of Face Detection

2.2 Offloading Tasks from Phone to Server

To measure the performance of face detection on real smartphone hardware, we implement a face detection application on an Apple iPhone 3GS using the OpenCV [5] library based on an existing implementation [6]. OpenCV is a widely used open-source library for computer vision and provides general-purpose object detection functionalities.

In addition to the standalone face detection on the iPhone, the application is capable of sending an image to a server and receiving detected results. The system diagram of the application is shown in Figure 2.

The *view controller* on the iPhone first takes an image from the *photo library* specified by the user and then gives the selected image to the *face detection client*. Based on a predetermined setting, the face detection client does either (1) local processing or (2) remote processing. In the case of (1) local processing, the face detection client processes the image using the local OpenCV library and gets the face detection results. In the case of (2) remote processing, the face detection client sends a face detection request to the server over a network (*e.g.*, WiFi or 3G) connection. Once the *face detection server* receives the request, it processes the image using the OpenCV library on the server and sends the result back to the face detection client on the iPhone. Finally, the view controller receives the detection results from the face detection client and visualizes the results as shown in Figure 1.

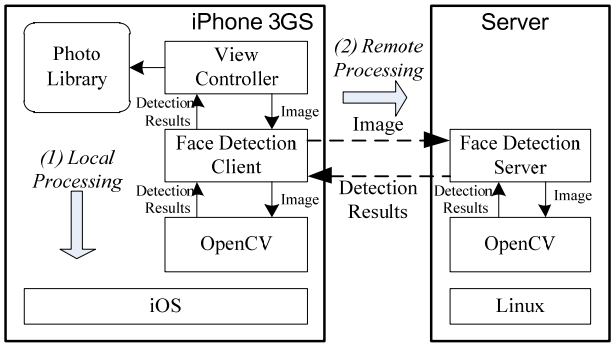

Figure 2. System Diagram of Application Task Offloading

2.3 Experimental Setup

In the experiment of face detection, we use an Apple iPhone 3GS (OS: iOS 4.2.1, detail specifications of CPU and memory have not been made public) and a server computer (CPU: Dual Core AMD Opteron Processor 870 2GHz, Memory: 32GB, OS: Linux 2.6.18.8). Both the iPhone and the server are connected in the same network segment via WiFi 802.1g.

The experiment tested 33 different images whose sizes range from 500K to 1.2 Mbytes. Those images contain at least a single face and at most eleven faces.

The version of OpenCV is 2.1.0, which is used in both the iPhone and the server side. The main function to detect faces is *cvHaarDetectObjects()* function provided by the OpenCV library, and the last four parameters given to the function are as follows: *scale_factor* = 1.1, *min_neighbors* = 3, *flags* = 0, *min_size* = cvSize(20,20).

During the experiment, no user applications on the iPhone other than the face detection application are launched. Bluetooth and 3G communication capabilities are turned off as well.

2.4 Speedup by Remote Processing

We tested the performance of both local and remote processing for the 33 images and computed speedup by the remote processing compared with the local processing. We show the result processing time in Figure 3 and speedup in Figure 4 respectively.

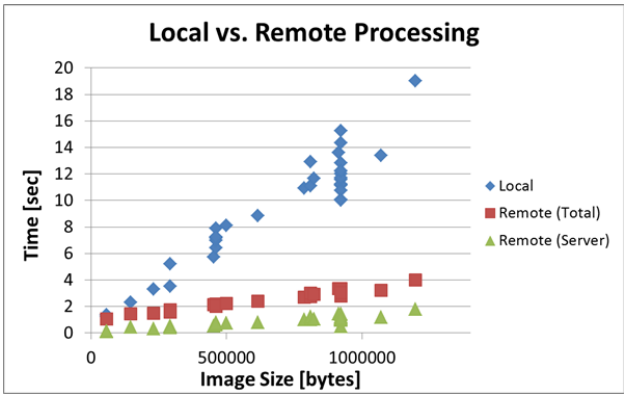

Figure 3. Performance of Local and Remote Processing

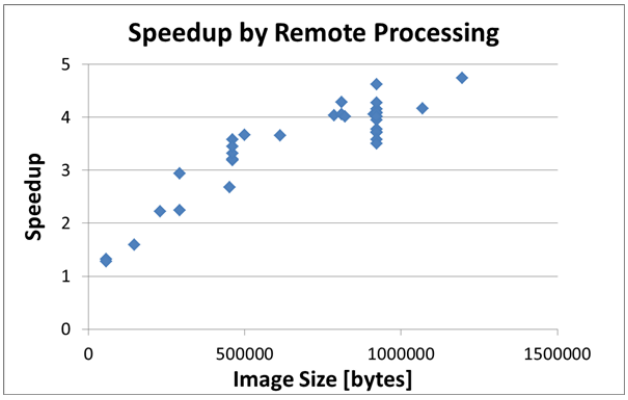

Figure 4. Speedup by Remote Processing

In Figure 3, the X-axis represents the size of each image, the plot *Local* represents the time required for local processing and the plot *Remote(Server)* represents the time required for the server to process an image when processing remotely. *Remote(Total)* represents the total time required for remote processing including Remote(Server) and the time for communication, i.e., sending the request from the iPhone and receiving the response from the server. From the graph, the remote processing is much faster than the local processing on the iPhone, and we can see the processing time of the images grows linearly for both local and remote processing.

Efficiency of the remote processing is clearly reflected in speedup as shown in Figure 4. On average, the remote processing is about 3.5 times faster than the local processing on the iPhone. An image of 1.2Mbytes improved the performance from 19 seconds to 4 seconds, which is the biggest improvement of all the images.

The network metrics over WiFi are RTT (Round-Trip Time) = 0.0142 seconds and bandwidth = 12.8701 Mbps at the time of the experiment. The performance of the remote processing totally depends on the network environment and the speed of the server, and it works effectively if both the network and the server are fast. In the real world, there are various combinations of networks and servers, and the condition of the network changes from time to time. Therefore, the question is when to process locally or remotely. To answer this question, we need a model that predicts the performance at run-time. Similar models can be developed for battery consumption.

3. DISTRIBUTED COMPUTATION MODEL

3.1 Remote Processing Model

We use the model depicted in Figure 5 to estimate total processing time when the client requests remote processing to the server. First, the user selects a problem to solve, and then the client takes it and sends a request with the problem to the server. Next, the server computes the received problem and responds a result to the client. Finally, the client shows the result to the user.

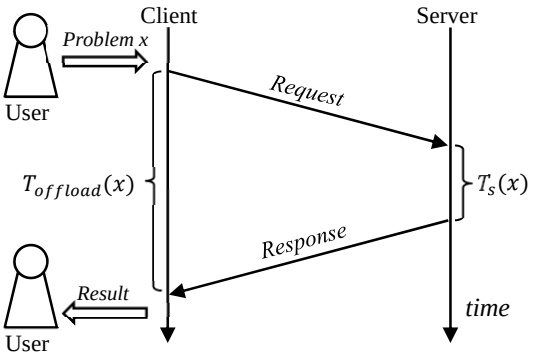

Figure 5. The Model for Remote Processing

Suppose the user selects a problem x, then the total processing time $T_{offload}(x)$ can be formulated as a function of x as follows:

$$T_{offload}(x) = T_{comm}(x) + T_s(x),$$

where $T_s(x)$ is the time required for the requested computation on the server and $T_{comm}(x)$ is the time required for sending the request and receiving the response.

Let $T_c(x)$ be the time required for the client to process problem x locally without help from the server; it is beneficial to offload the problem to the server clearly if $T_{offload}(x) < T_c(x)$.

The assumption here is that the client and the server know about basic equations of $T_{comm}(x)$, $T_s(x)$, and $T_c(x)$ for a specific problem, but do not know about the parameters of the equations initially. This is because the client is a mobile device, and the user may use it on different networks and connect it with different servers. Moreover, because applications running on the client or the server can be downloadable on devices with various hardware settings, it is fair to assume that the applications do not know how fast the devices are until they actually run a problem on particular hardware.

3.2 Parameter Update Protocol

The client and the server communicate to update their internal parameters for a better prediction of computation time. To predict and compare both the local and remote processing time, we need to compute problems both locally and remotely for the first execution. From the second execution, the model predicts and compares the performance based on the parameters before running the computation, and updates the parameters accordingly from computation results. A protocol and a method to update those parameters are explained in order.

Update Protocol for $T_s(x)$ and $T_{comm}(x)$: We update $T_s(x)$ and $T_{comm}(x)$ as shown in Figure 6. First, the client sends a process

request x' to the server, and the server processes it and measures the process time as $T_s' = T_s(x)$. Using a (x', T_s') pair and an approximation method such as least squares, the server updates $T_s(x)$ to approximate a set of measured data observed in the server. Next, the server responds with the computed result of x' back to the client as well as measured T_s' and updated $T_s(x)$. Once the client receives those parameters, then it calculates T_{comm}' by subtracting T_s' from the measured $T_{offload}'$ by the client. Finally, the client updates its $T_{comm}(x)$ by T_{comm}' and make a local copy from the received $T_s(x)$.

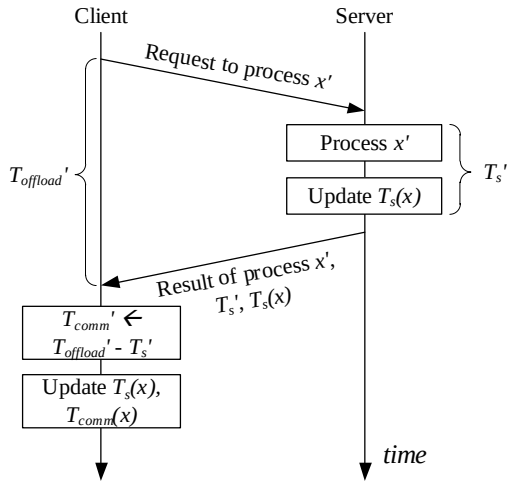

Figure 6. Update Protocol for $T_s(x)$ and $T_{comm}(x)$

Update Method for T_C: For the first execution of the problem or in the case $T_{offload}(x) > T_c(x)$ is true, the client locally processes the problem x'. It also measures the time required for the local process as T_c'. Just as the server updates $T_s(x)$, it updates $T_c(x)$ using a (x', T_c') pair to approximate the measured computation time.

Apart from the processing time itself, the protocol and method to update parameters are light-weight so that they are able to apply at run-time without serious overhead. If a prediction error either for $T_{offload}$ or T_c becomes less than a certain threshold (*e.g.*, 10%), the above update protocols pause since it can be considered that the prediction works well enough, therefore there is no need to update the parameters. Conversely, the protocols resume if the prediction error (measured periodically with progressively less frequency) becomes greater than a certain threshold. When resuming, all the previous trained data will be reset to adapt to the new environment.

4. MODEL INSTANTIATION

We choose the face detection as an application of the model described in the previous section. Table 1 shows the relationship between the parameters used in the model and their instantiation in the face detection problem.

Table 1. Model Instantiation

Model	*Instance*
x	An image
Processing x	Detecting face regions from an image x
$T_s(x), T_c(x)$ and $T_{comm}(x)$	Linear functions of the size of an image x
Update method for $T_s(x), T_c(x)$ and $T_{comm}(x)$	Least squares method

The reason why we choose linear functions as the models for $T_s(x)$, $T_c(x)$, and $T_{comm}(x)$ is that for this problem we know those functions to be linear as confirmed by preliminary experiments reported in Section 2. Appropriate functional models and corresponding approximation methods for updating the functions are necessary for this model to work properly.

5. EXPERIMENTAL RESULTS

In this section, we present a detailed evaluation of our model implementation by predicting the remote processing time of the face detection task. We use the model instantiation described in Section 4, and we can find out how well the model predicts the remote processing time when we pick images from the 33 images used in the experiment in Section 2.

We perform the experiment with three different cases: 1) The network and server are stable; 2) The network becomes temporarily slow; and 3) The server becomes temporarily busy. For each case, we tested ten randomly selected sequences each consists of 33 images and measured the average time of remote processing ($T_{offload}$) and compare it with the predicted value.

Case 1 - Network and Server Stable: There is no dynamic disturbance from the environment in this scenario, that is, no other clients use the network and server. Figure 7 shows the prediction result of $T_{offload}$. As we can see from the graph, the model predicts $T_{offload}$ well. The parameters used in this prediction are generated by a relatively small number of trials as shown in Figure 8. In the training stage, the model generates somewhat erroneous predictions, but these prediction errors are unavoidable since there are not sufficient learning examples available at the early stage of the training. However, the least squares method works better as the trials progress. The error becomes below 10% threshold after 7.8 trials on average, and the training becomes inactive afterwards. During the inactive period of the training, the prediction error stays below 10 %. The reason why the model predicted $T_{offload}$ well is that $T_{offload}$, T_{comm}, and T_s are linear functions, therefore, the least squares method works well.

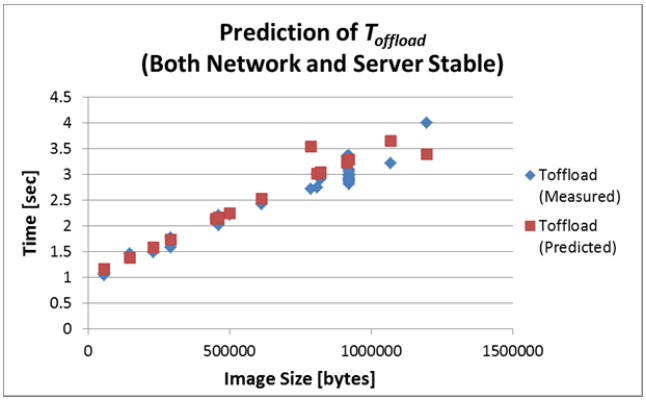

Figure 7. Prediction of $T_{offload}$ (Both Network and Server Stable)

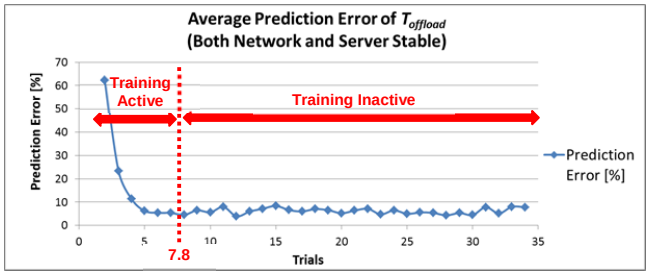

Figure 8. Average Prediction Error of $T_{offload}$ (Both Network and Server Stable)

Case 2 - Network Temporarily Slow: In this scenario, the network becomes twice slower in the middle of the trials (from the 10th to 20th) than in the rest of the trials. Figure 9 shows the prediction result of $T_{offload}$. The plots spread wider than Case 1 due to the disturbance, however, the model predicts $T_{offload}$ relatively well as we can see both plots for the measured and predicted are close in the graph.

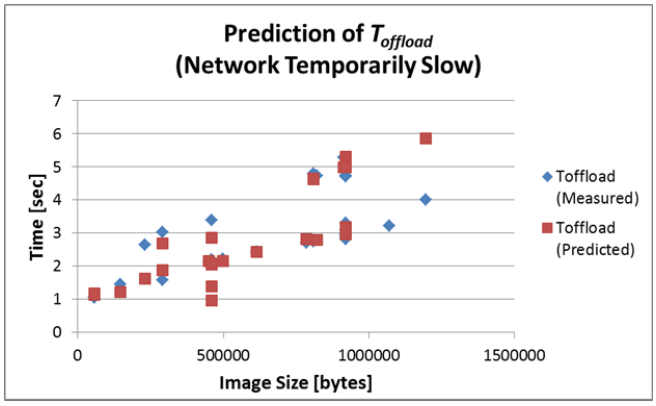

Figure 9. Prediction of $T_{offload}$ (Network Temporarily Slow)

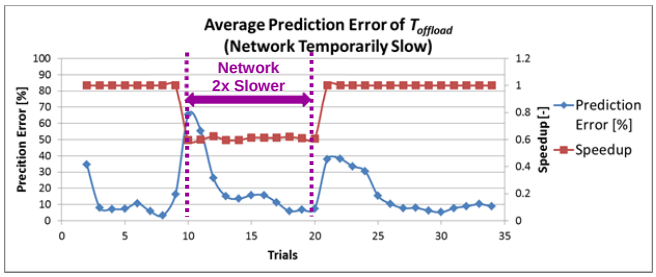

Figure 10. Average Prediction Error of $T_{offload}$ (Network Temporarily Slow)

In Figure 10, the prediction error increases quickly at the 10th trial and gradually decreases as the trial progress until it rises up again at the 21st trial. After that, the error gradually goes down again. Overall, the parameter update protocol works adaptively to the network speed change.

Case 3 - Server Temporarily Busy: In this scenario, the server becomes busy and twice slower in the middle of the trials (from the 10th to 20th) than in the rest of the trials. Figure 11 shows the prediction result of $T_{offload}$. Same as Case 2, the plots spread slightly wider than Case 1; however, the model predicts $T_{offload}$ well despite the server overload.

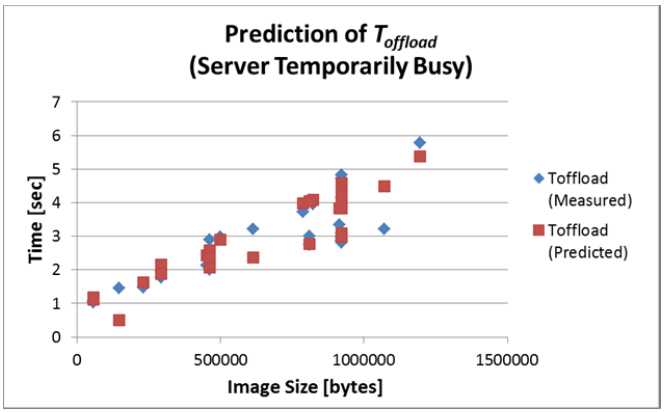

Figure 11. Prediction Error of $T_{offload}$ (Server Temporarily Busy)

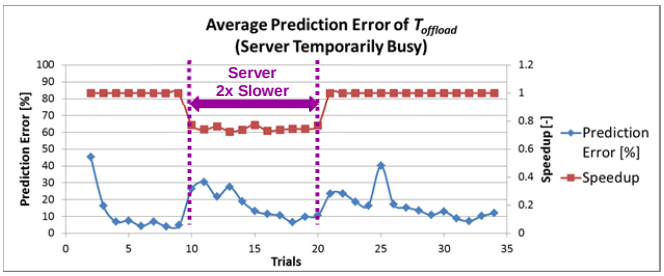

Figure 12. Average Prediction Error of $T_{offload}$ (Server Temporarily Busy)

Same as in the network speed change, we can see that the client can follow the server speed degradation as shown in Figure 12, however, the peaks of the prediction errors are lower than Case 2. This happens because the communication takes more time than processing at the server, so the effect for overall time is limited relative to the network speed change.

6. RELATED WORK

Wang and Li [1] introduce a compiler-based tool that partitions a regular program into a client-server program. MAUI[4] lets developers specify which methods of a program can be offloaded for remote execution, focusing on minimizing battery consumption.

Cyber foraging [7] uses nearby surrogate computers to offload the computational burden from mobile devices. Similarly, Slingshot [8] replicates a remote application state on nearby surrogate computers co-located with wireless access points to improve performance. These approaches are similar to ours in a sense that they assume applications are partitioned before execution and utilize nearby computers, but they do not have an explicit mechanism to adapt to the environment changes.

There are approaches using virtual machine (VM) migration including Cloudlet [2] and CloneCloud [3]. Application developers do not need to modify applications, but both approaches require transferring the entire or partial VM image from the smartphone to the cloud as well as complex profiling of

the remote execution environment (*e.g.*, network/cloud performance) at run-time.

7. FUTURE WORK

In the future, we plan to apply our method to other applications to confirm the generality of the method. Candidate applications include but not limited to real-time face detection for video and real-time rendering of CFD (Computational Flow Dynamics). The face detection for video is a natural enhancement of the example we have done this time and is also expected to be pretty computationally intensive since we need to detect faces at very high frequency. Face recognition (comparing detected faces to a DB of images) is even more intensive in terms of computation and data requirements, and therefore it requires our task offloading approach to be practical. CFD is known as one of the most computationally intensive applications in general. Running CFD on smartphones sounds too complex, but it might be useful for games in terms of creating realistic motion of fluids and interaction with them.

Another direction of enhancement could be on the server side. If an nVidia's or an AMD's graphics card is available on a server computer, we could utilize the GPUs by using CUDA or OpenCL and speed the process up greatly. We could also utilize tablets, such as Apple's iPad and Android Tablets, as servers since nowadays they use fast multicore processors. Offloading can also occur from a tablet to a server or a cloud. Cloud computing is suitable for task offloading purpose as well because it can provide great scalability in performance and give virtually infinite computing resources. Latency between a smartphone and clouds would vary depending on the location of the smartphone; therefore, light-weight adaptation on the smartphone shown in this paper would be critical.

Using an actor-oriented programming language such as SALSA [9] is one way of implementing dynamically reconfigurable smartphone applications. In SALSA programming, an actor is the unit of execution. Consequently, an application written in SALSA is natively partitioned as a collection of actors and is suitable for dynamic partioning of the application at run-time. One research direction is to enhance the model presented in this paper to support dynamic partioning using SALSA while keeping the model light-weight.

8. CONCLUSIONS

We have described a model to predict the total processing time when offloading part of an application from smartphones to servers. In our model, if an application developer can (1) define a basic model of the problem (*e.g.*, $f(x)=ax+b$) and (2) implement an algorithm to update the model (*e.g.*, least squares method), the application quickly adjusts the parameters of the model and minimizes the difference between predicted and measured performance adaptively. We also assume that an application developer statically partitions the program for the remote execution, however, this model's simplicity greatly reduces the time required for profiling the performance of the application at run-time and thus enables users to start using an application without pre-computing a performance profile. The model is applicable not only to nearby servers, but to servers in distant locations. However, it is practical and effective when used with nearby servers due to its low-latency network environment.

The experiments have shown that our update parameter protocol for the performance prediction functions works sufficiently well for the face detection problem. It takes some time to adapt to the network or the server performance change, but the prediction error gradually becomes smaller and finally a state is reached where a parameter update is no longer required.

To use our method, the server program has to be downloaded to a computer before the user starts using the application on the smartphone. Since the proposed task-offloading method is adaptive as shown in the experiments, we can use newer/older computers and faster/slower networks, yet guaranteeing a higher (or at least no lower) quality of service to smartphone users at home. Therefore, even slow computers on fast home networks could help improve the user experience on smartphones.

ACKNOWLEDGEMENT

We would like to thank Gustavo A. Guevara S. and Qingling Wang for their valuable feedback. We would also like to express our gratitude to Evan Patton who kindly guides us how to setup the iPhone development environment. This research is partially supported by NSF CAREER CNS Award No. 0448407 and by the Air Force Office of Scientific Research.

REFERENCES

[1] Wang, C. and Li, Z. A computation offloading scheme on handheld devices. *In Proceedings of J. Parallel and Distributed Computing*. 2004, 740-746.

[2] Satyanarayanan, M., Bahl, P., Cáceres, R., and Davies, N. The Case for VM-Based Cloudlets in Mobile Computing. *In Proceedings of IEEE Pervasive Computing*. 2009, 14-23.

[3] Chun, B., Ihm, S., Maniatis, P., Naik, M., and Patti, A. CloneCloud: elastic execution between mobile device and cloud. *In Proceedings of EuroSys*. 2011, 301-314.

[4] Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. MAUI: making smartphones last longer with code offload. *In Proceedings of MobiSys*. 2010, 49-62.

[5] OpenCV, http://opencv.willowgarage.com/wiki/

[6] Niwa, Y., Using OpenCV on iPhone, http://niw.at/articles/2009/03/14/using-opencv-on-iphone/en

[7] Balan, R.K., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H. The case for cyber foraging. *In Proceedings of ACM SIGOPS European Workshop*. 2002, 87-92.

[8] Su, Y. and Flinn, J. Slingshot: deploying stateful services in wireless hotspots. *In Proceedings of MobiSys*. 2005, 79-92.

[9] Varela, C.A. and Agha, G. Programming Dynamically Reconfigurable Open systems with SALSA. *In Proceedings of SIGPLAN Notices*. 2001, 20-34.