Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 11-006

Dynamic Outsourcing Mobile Computation to the Cloud

Chonglei Mei, James Shimek, Chenyu Wang, Abhishek Chandra, and
Jon Weissman

March 14, 2011

Dynamic Outsourcing Mobile Computation to the Cloud

Chonglei Mei, James Shimek, Chenyu Wang, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{chomei,shim0125,chwang,chandra,jon}@cs.umn.edu

March 23, 2011

Abstract

Mobile devices are becoming the universal interface to online services and cloud computing applications. Since mobile phones have limited computing power and battery life, there is a potential to migrate computation intensive application components to external computing resources. The Cloud is an attractive platform for offloading due to elastic resource provisioning and the ability to support large scale service deployment. In this paper,we discuss the potential for offloading mobile computation for different computation patterns and analyze their tradeoffs. Experiments show that the we can achieve a 27 fold speedup for an image manipulation application and 1.47 fold speedup for a face recognition application. In addition, this outsourcing model can also result in power saving depending on the computation pattern, offloading configuration, and execution environment.

1 Introduction

Today, mobile devices, such as smart phones, tablets, PDAs, and ipods, have become indispensable in our daily lives. With their growing popularity, users have come to rely more and more on them as their primary computation and communication devices, and are even beginning to expect functionality and performance similar to that from traditional computing devices such as desktops and workstations. However, meeting such expectations on these devices is challenging due to several reasons. First, the mobility of these devices implies that they are battery-powered, limiting their power capabilities. As the devices become more popular and their use becomes more frequent, the increasing energy consumption becomes a key bottleneck in their computational capability. Second, the computational power, in terms of processing and memory, is severely limited compared to traditional computers. Thus, these devices are limited in their ability to execute rich user applications involving extensive use of computer vision and graphics, speech recognition, or machine learning, which can be resource intensive.

To overcome these constraints of mobile devices, external resources can be introduced as an extension to the capabilities of these devices [16, 11]. Computation-intensive or power-intensive applications can then be outsourced to the external computing resources. This approach can potentially improve the performance of mobile applications, enable mobile devices to support applications that current devices cannot even execute due to their resource constraints, and save precious energy as well. While one approach to acquire such external resources may be via fixed resources (e.g., static servers, proxies), this approach has several limitations. First, many mobile applications have properties of large resource sharing and high mobility, e.g., location based online mobile games, so that static resources may not be able to support large-scale backend computational requirements in a locality-aware manner. Second, the usage profiles of many applications are highly dynamic, varying with user interest and time-of-day, so that static allocation of resources could result in poor availability during times of high demand and resource wastage at other times.

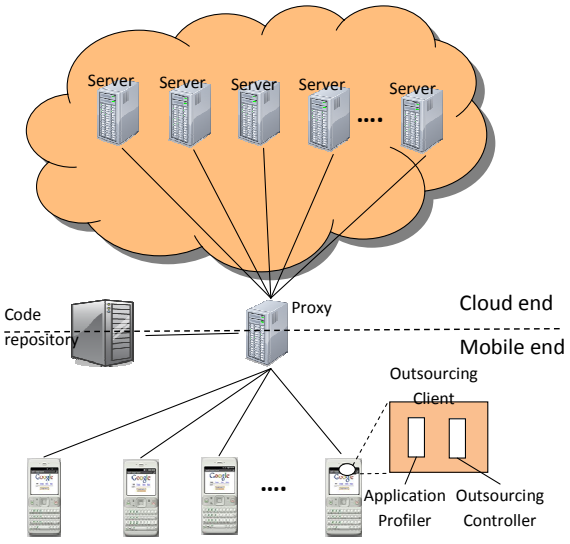


Figure 1: Mobile computation outsourcing framework

To overcome these limitations, we propose the use of a cloud as a backend to outsource mobile computations. A public cloud, such as Amazon EC2, can provide elastic and "unlimited" computation and storage resources. Thus, it can adjust the amount of resources according to the service requests and provide large-scale deployment easily. It can also enable easy data and compute sharing among multiple devices interacting with each other or through the same application. However, a cloud may be limited by its connectivity to the mobile devices and its benefit is likely to be highly dependent on the computation-communication tradeoff in an application [14].

In this paper, we identify the opportunities and tradeoffs of using the cloud as a mobile outsourcing backend. As part of this work, we have created a prototype outsourcing framework based on the Android platform and Amazon EC2. Using this framework, we have studied the outsourcing potential of different applications by profiling their behavior under three computing environments—local execution on the mobile device, and hybrid execution with the mobile device and the remote cloud through a WIFI and 3G network respectively. Under each case, computational resource usages (i.e., CPU, memory and power), and execution time, are monitored and analyzed. Our results provide insights into the feasibility and tradeoffs of outsourcing based on application and environmental characteristics. In addition, we also propose adaptive policies that can enable the outsourcing decision of an application based on the profiling results, available resources, and user preferences.

2 Outsourcing Framework and Profiling Methodology

2.1 Mobile Computation Outsourcing Framework

In order to outsource computation from mobile devices to external computing resources, a mobile computation offloading framework is proposed as shown in Figure 1. This framework is implemented in Java, executes on the Android mobile platform [2] and can be deployed easily on any backend platform.

There are three components at the cloud end: the proxy, code repository and server. The proxy provides a gateway between the mobile device and the cloud backend. In some cases, the proxy could be replicated in the cloud for scalability, and in others, the proxy could be local to the device and outside the cloud. The proxies are configured to have access to a code repository which contains popular code components that may be launched on a cloud server. The mobile device connects to

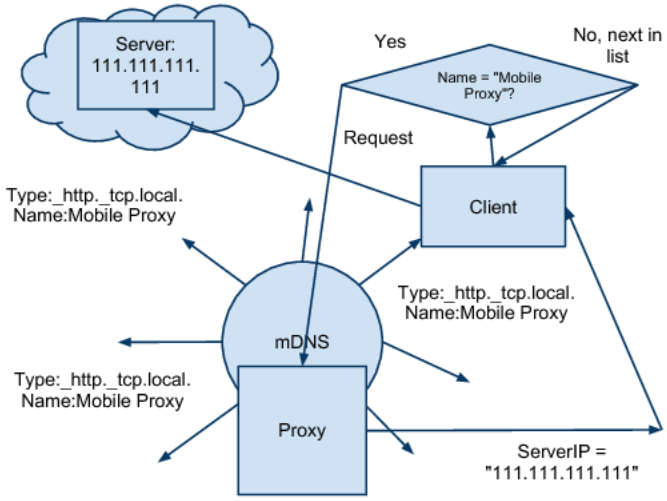


Figure 2: Local Service Discovery

a proxy to initiate outsourcing by naming a specific Java class component. There are three cases: the code is already running in the cloud, the code is available in the repository but not deployed, or the code is available only on the mobile and must be uploaded by the mobile device. After the code location and the mobile server are identified, the mobile device will communicate with the mobile server directly. So the proxy is not on the data communication path, which makes the proxy not a bottleneck. This process is shown in Figure 2.

If the mobile proxy is out of the cloud, it can run mDNS service to broadcast the mobile outsourcing service. Thus, it works as an access point for the mobile devices. We can distribute proxies as WiFi access points to provide outsourcing service. The service discovery is shown in Figure 2. The proxy advertises itself through mDNS service. The advitised information includes the service type(http,tcp,local,etc.), and the service name(i.e., Mobile Proxy). The mobile client has no fore knowledge about this proxy. It searches for advertised service on the local network(or Internet, if the service is registered). If the proxy is found, the client connects to the proxy based on the information contained within the broadcasting information. After that, then proxy will send back the addresses of the actual mobile server to the mobile user. Then the mobile user can connect to the server for following services. In the current implementation, we use the Java implementation, JmDNS, which is very light weight.

If the mobile proxy is installed within the cloud, some other approaches are needed to make the proxy known to mobile devices. In such case, the mobile proxy can be associated with some well known URL. Then the mobile devices can connect to the URL to obtain computation outsourcing services.

Each mobile server is used to hold services for some mobile devices. The mobile proxy can start the mobile server dynamically when new requests are received. The mobile proxy has the information of mobile servers that it manages. It can make decision on where to start the server, i.e., to start a new server or deploy the service on a running server.

Since the mobile devices and the mobile server may run different versions of code, it is necessary to have the code repository to store the source code of popular service components for the mobile server. The code repository can be located either in or out of the cloud. For optimization purpose, it can be set within the cloud close to the mobile servers to decrease the access time of the source code.

An outsourcing client at the mobile end is used to manage computation offloading. The ap-

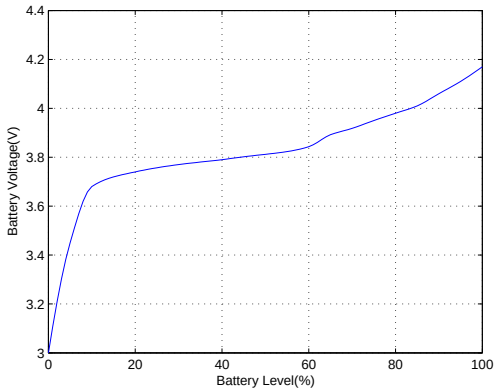


Figure 3: Battery Voltage V.S. Battery Level

plication profiler monitors the available resources on the mobile device, including CPU, memory, battery usage, and network status, and records the performance history of offloaded application components. The outsourcing controller has two responsibilities. First, it makes outsourcing decisions based on the available resources and profiling data. Second, it triggers outsourcing at the appropriate time.

When the outsourcing client communicates with the mobile proxy, a class, called *Command*, is used to simplify the communication interface. A *Command* class includes the following fields:

ClassName	The package name that the server needs to load
Operation	The operation that needs to perform based on the class
Payload	Additional assisting information

When the mobile proxy receives a Command, it analyzes the the required Java class based on ClassName and starts the service at an available server.

In the current implementation, since the code repository and the server are located at the same machine, all testing codes are stored in a fixed directory. The mobile proxy loads the required Java class dynamically based on the ClassName, which is a running server. Then the server receives following commands from the mobile client to perform computation, e.g., the computation data is sent to the server through the *Payload* field of *Command*.

2.2 Profiling Methodology

The decision to outsource will be dependent both on the application characteristics (e.g., its computational requirements) and the current environment (e.g., the amount of remaining battery, network connectivity, etc.). Thus, in order to make the right outsourcing decision, the application's performance and resource usage needs to be profiled under different environments. In our current implementation, five parameters are monitored and analyzed in different execution configurations, i.e., input data size, CPU usage, memory usage, execution time and power consumption. Three execution modes are considered: i.e., **Local** (executed on the mobile), **Remote-WIFI** (outsource to the remote server through WIFI), and **Remote-3G** (Outsource to the remote server through Sprint 3G network).

The system file '/proc/stat' and the Debug class [3] can provide CPU and memory usage for each process in the Android system. If the execution mode is Local, the total execution time is recorded. If the execution mode is Remote-WIFI or Remote-3G, the total execution time (including communication time), server processing time, and communication time are recorded separately.

The Android system estimates the battery level based on the battery voltage and broadcasts the

battery level information to all processes. However, there are two drawbacks with this approach. First, it works at a coarse granularity, as it only updates the battery information when it changes by 1%. Secondly, the Android system estimates the battery capacity based on the battery voltage, which is affected by many factors, e.g., the temperature and the discharge current. Experiments show that the voltage fluctuates with the discharge current and temperature. Thus, when the mobile device is running, it is inaccurate to rely on the battery information provided by the Android system.

The approach we adopted in this paper is to estimate the battery capacity using the voltage-battery level curve as shown in Figure 3, which is generated based on the open-circuit voltage at 60^oF. As different batteries may have different voltage-battery level curves, Figure 3 only applies to the battery we used in this paper. When monitoring the power usage, the open-circuit voltage of the mobile battery is recorded before and after the application execution, denoted as V_{start} and V_{end}. Then they are converted to the battery level according to this curve as L_{start} and L_{end}. Thus, the power consumed by the execution is $(L_{start} - L_{end})$.

3 Tradeoffs in Offloading to the Cloud

In this section, we analyze the impact of different application patterns on performance and energy tradeoffs, by offloading computation to the Amazon EC2 cloud through our outsourcing framework. Two applications are analyzed, image manipulation and face recognition. Profiling statistics are used to show the offloading tradeoffs and how offloading decisions can be made dynamically. We used an HTC hero (cpu 528 MHz, memory 200MB running Android 2.1) and installed the backend server on a small EC2 instance (1 EC2 Compute Unit and 1.7GB memory). The average network latency to access EC2 over WIFI and Sprint 3G was 82ms and 151ms respectively(compared to 44ms from a wired machine).

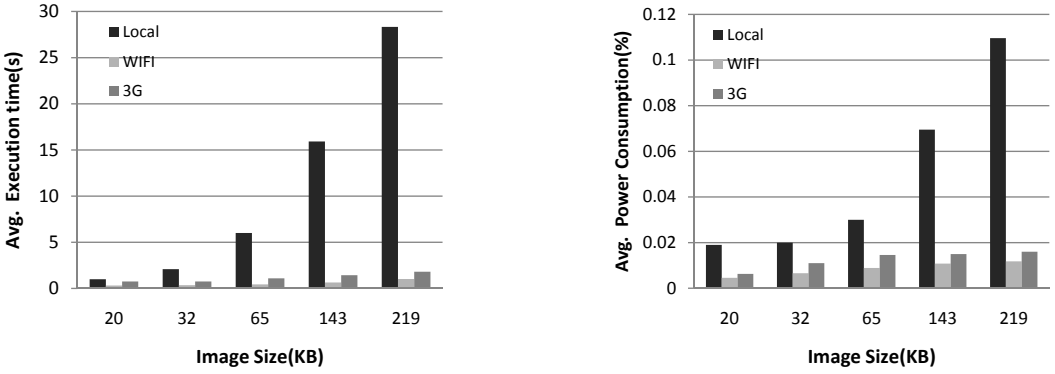
3.1 Applications

Image manipulation is an important application category, especially as most mobile devices are equipped with high-resolution cameras. Users can edit, or add visual effects by applying different processing filters [4] on the image. The performance of four filters are compared in this section: SimpleBlur, SimpleSharpen, GrayScale, and TileScale.

SimpleBlur and SimpleSharpen filters perform image convolution on the original image. The GrayScale filter converts a color image into a gray scale image. The TileScale filter tiles a smaller scale of the original image on the original image canvas. Of these filters, SimpleBlur and SimpleSharpen have similar performance and are compute intensive, while GrayScale and TileScale are not. In our results, we use SimpleBlur and GrayScale as a representative for each class (compute intensive vs. compute non-intensive). As the amount of computation is determined by the input image size, six different JPEG images with the same compression ratio are compared, i.e., 20K, 32K, 65K, 143K, 219K, and 300K. When the image size is over 300K, SimpleBlur on the mobile cannot finish due to heap memory overflow. Thus, such cases are not presented in our results, however, these cases present another motivation for offloading by overcoming memory constraints.

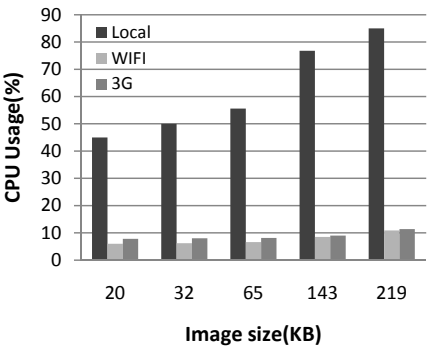
Face Recognition is another useful application on the mobile device. In this experiment, a Haar-Classifier [1] algorithm is used. It applies a series of simple feature detectors of different sizes to an image. Each feature detector is thresholded and the outputs are combined in a manner specified by the Haar cascade to yield a boolean output indicating whether the face is in the image.

These two sets of applications also differ in their communication patterns. For image manipulation, the user may apply multiple filters on the same image, thus obviating the need to resend the source image. On the other hand, with face recognition, the user would normally process different images, which requires the mobile device to upload a separate image every time.

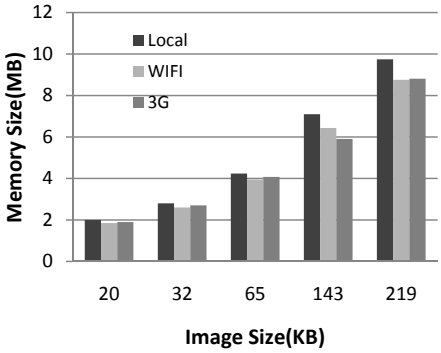


(a) Average Execution Time

(b) Average Power Consumption



(c) CPU Usage



(d) Memory Usage

Figure 4: Statistics of SimpleBlur

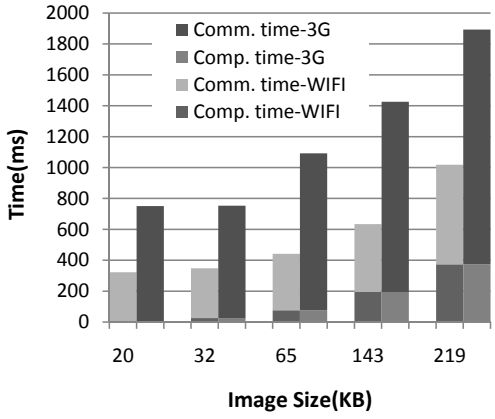
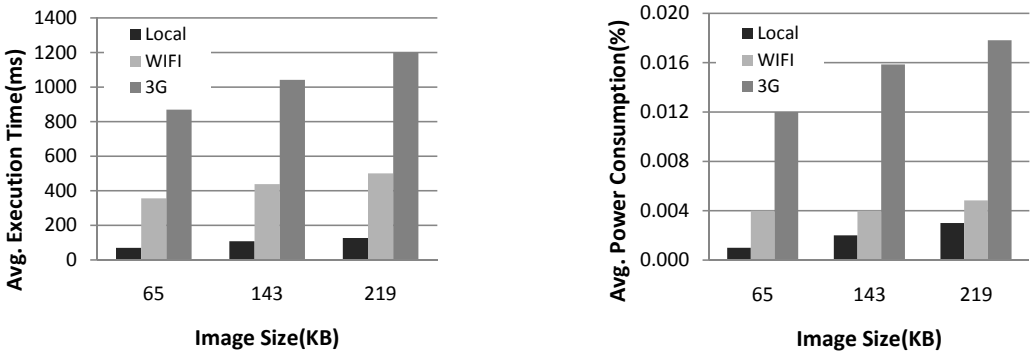


Figure 5: Remote execution time breakup for SimpleBlur



(a) Average Execution Time

(b) Average Power Consumption

Figure 6: Statistics of GrayScale

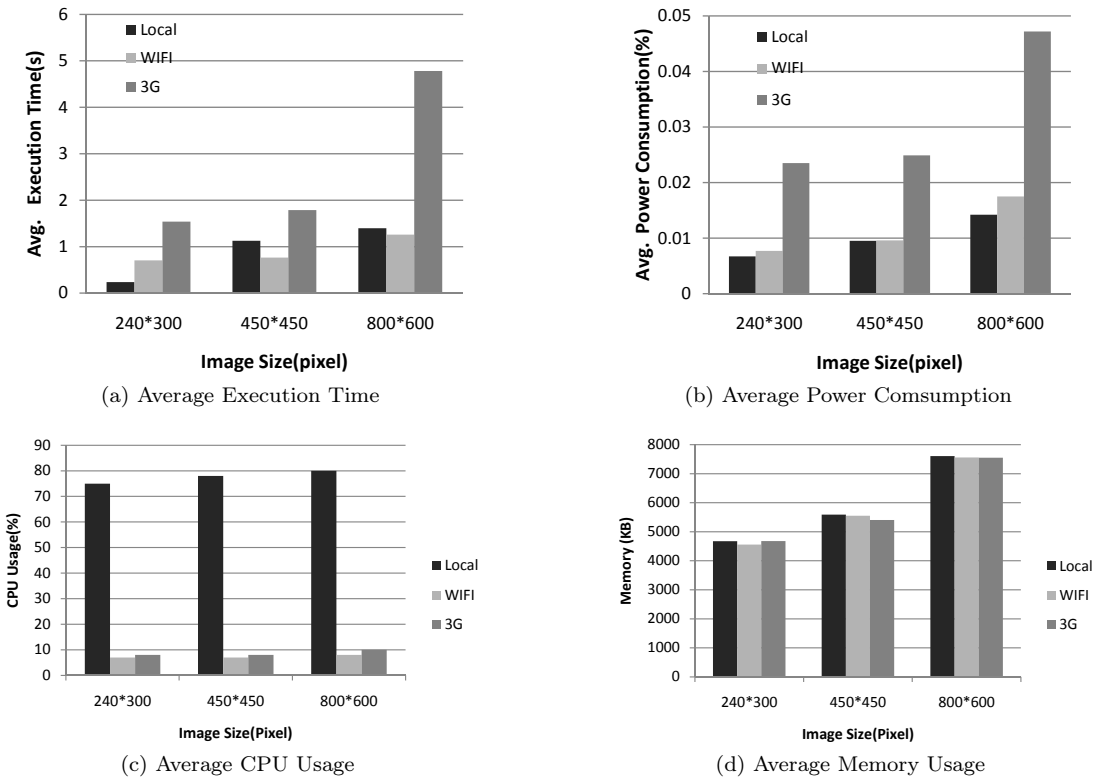
3.2 Profiling Results

The profiling statistics for the SimpleBlur filter are shown in Figure 4. Figure 4a shows the average execution time per run. Here, for the remote execution case, the execution time is end-to-end, including the network communication time as well as the backend computation time. As seen from the figure, for all image sizes, the local execution mode spends the most time and Remote-WIFI spends the least. Since the 3G network is slower than the WIFI network, the Remote-3G mode is slower. The compute vs. communication time break down for the remote execution modes is shown in Figure 5. It shows that the remote execution time is dominated by the communication time. The greatest speed gain is 27.7× when the image size is 219K with WIFI. The results also show that the speed gain increases proportional to the input image size.

Figure 4b shows the average power consumption (i.e., percentage of the total battery capacity) per run. The results show that for each image size, the local execution consumes the most power and the Remote-WIFI consumes the least. When the image size is $219K$, local execution consumes 9.28 times the power of Remote-WIFI. This result can be attributed to the high computation requirement of the SimpleBlur filter, as shown in Figure 4c, which plots the CPU usage on the mobile device for different image sizes. It shows that when the SimpleBlur is executed locally, the CPU usage is over 50% and it increases with image size. When SimpleBlur is offloaded to the remote server, the CPU usage drops to below 10%. We also found that for an image size, all three execution modes consume nearly the same amount of memory. This is because the mobile device has to maintain almost the same amount of buffers.

Figures 6a and 6b show the execution time and energy usage respectively for the GrayScale filter. Since it has similar statistical features of CPU and Memory usage as SimpleBlur, these two statistics are omitted here. In this case, we observe that when offloading computation to the Cloud, it takes more time and consumes more power than the local execution for all image sizes. The reason is that this filter has fairly light-weight computational requirements, so that the communication overhead is the dominant factor in performance as well as energy usage.

For face recognition, the relation between execution time and power consumption demonstrate different patterns as shown in Figure 7. For all three modes, Remote-3G takes more time and consumes more power than the local mode. Thus, it may not be appropriate to use 3G network to outsource face recognition operation (if the goal is to optimize this application only). Figure 7a shows that when the image is small (240×300), Remote-WIFI costs almost the same amount of power as the local version. However, when the image size increases ($\geq 450 \times 450$), Remote-WIFI has better performance. Experiments show that when the image is larger than 240×300 for Remote-WIFI, 50% more power is consumed, but the response time can be reduced by 1.47 times. The CPU usage and memory usage are shown in Figure7c and Figure 7d. We can see that they have similar patterns with image manipulation. Thus, it is good to offload the Face Recognition to



(a) Average Execution Time

(b) Average Power Comsumption

(c) Average CPU Usage

(d) Average Memory Usage

Figure 7: Statistics of Face Recognition

the cloud to achieve multi-tasking features.

One outsourcing overhead not included in our results above is the virtual machine startup time on the Amazon EC2 platform. If a new server is needed, it takes around 40 seconds to start up. However, this is just a one-time cost when a server is started, and once the server is up, it can handle multiple requests.

4 Discussion of Outsourcing Strategy

Our results highlight three different application patterns that impact the outsourcing strategy. SimpleBlur and SimpleSharpen are computation intensive, which gains significant speedup and energy saving through computation outsourcing. TileFilter and GrayFiler are computation non-intensive, which gains no performance or energy improvement. Face Recognition demonstrates a case where offloading benefits depends on the execution environment (input data size and network connectivity). Thus, the decision to outsource depends heavily on the application characteristics and the execution environment.

The offloading decision can be made based on the user's preference, i.e., low response time, long battery life, or both. A group of pre-defined preference-policies can be stored at the mobile device which can be selected by the user. Preference-policies include the desired operating range threshold for each metric (CPU usage, response time, power consumption, etc.) generated based on the profiling data. For instance, if the users prefer to run more applications, then applications should offload as many operations as possible to the Cloud. If the user prefers long battery life or low response time, the mobile can make decision based on the preference-policy and the available resources. After the operation is offloaded to the remote server, the outsourcing controller will monitor the execution and adjust the offloading accordingly. We are building such a system.

In some cases, outsourcing may be necessary simply to make an application execution feasible. For instance, the Android system has an upper limit of heap memory for each application, e.g., 24M for the Nexus One, so that it cannot execute applications consuming memory over this limit. In our experiments, an image over 300K cannot be processed by SimpleBlur locally. Thus, offloading such memory-intensive applications to the Cloud can extend the computability of mobile devices.

Besides the performance and energy benefits to the application being offloaded, outsourcing can also benefit other applications running on the mobile. As profiling statistics show, computation outsourcing reduces the CPU usage to below 10% for all profiled operations. Thus it frees the CPU time for more applications, so that, multi-tasking performance of the mobile can be enhanced by computation outsourcing.

The remote execution time is determined by the network status, i.e., a slower network result in longer execution time, which again leads to high power consumption. For each input size, there should be a theoretical threshold time, $T_{threshold}$, at which the local execution and the remote execution consumes the same amount of power. When making the offloading decisions, $T_{threshold}$ can be used to determine whether the offloading is energy efficient. Let t be the predicted execution time for the remote execution, and T_{local} the local execution time, there is the following scheduling rule, if $t < T_{threshold}$, outsourcing saves energy; if $t < T_{local,}$, outsourcing gains speedup.

There are other potential benefits to using the Cloud as the backend. Common application components may already be running on behalf of other mobile users, and collaboration between different services in the Cloud will improve outsourcing efficiency. Since multiple applications will place some of their components and data in the Cloud, there is potential to implicitly share these code and data. For instance, the face recognition application might need image manipulation to preprocess the image. Thus, it can share the same image manipulation server as other users.

5 Related Work

Offloading mobile computation to local servers has been proposed by others. In some projects, static decisions are made before application execution, e.g. Spectra [5, 7], Calling the cloud [8], and [10], in contrast to our dynamic approach. Other projects have explored parts of the space, [14] studies just the energy-savings for different communication patterns, but no implementation is provided. Similarly, [11] analyzes potential energy savings but no implementation is provided. In [9], bandwidth and memory are used as the decision-making factors, but CPU and battery life are not considered. Wishbone [15] proposes an approach to partition mobile sensor applications based on network and CPU, but not the battery budget. In contrast, we collect sufficient profile information to enable more holistic decisions.

Other projects are based on virtualization. [6]present the idea of full VM cloning of a mobile image to a remote server at a coarse-grain level, but no implementations are evaluated. Cloudlets [16] propose to use local servers as a cloud target. Both the cloudlet server and mobile device run virtual machines, while our framework exploits the sharing and scale potential of the commercial cloud. Protium [12] addresses the issue of session mobility and universal data access from any client. Slingshot [17] evaluated the use of a surrogate running at a hotspot coupled with a remote server. The idea is to use the hotspot for outsourcing when Internet connections between it and the server are poor, but to use the remote server when the user is moving and connection to the hotspot is lost. Our approach is more general and considers a wider-range of resource availability beyond networking.

Compiler-assisted approaches have also been proposed [13, 18, 10]. These works are limited either to static mapping decisions or single application optimization only. In [18] dynamic placement is considered but only the metric of energy is used and is limited to single applications. Diet[10] uses a Java-based program analyzer which can partition Java programs at the method-level. Unlike our work, they are limited to single applications and are solely on performance.

6 Conclusion and Future Work

In this paper, we have shown the potential of the cloud for dynamic mobile outsourcing, and the potential tradeoffs and limitations. We are unaware of any other live experimental analysis to date. The results show that offloading compute-intensive applications to the cloudte-intensive applications to the cloud can greatly reduce the response time and battery consumption. With WIFI network, it can achieve more than a 27× speedup and 9× power efficiency. Future work is focused on using the profile information automatically at run-time to make offloading decisions.

In the future, an execution scheduling system can be built to select the execution plan with the best user experience given the available computing resources. The system can make decisions dynamically and deploy the services. The user can define their own preferences as well, e.g., save power, improve performance or both. Since the outsourcing framework is much more computationally powerful, the mobile client and the cloud server can work on different data sets for the fidelity concern. For instance, the face recognition application can perform on a coarse grain scale while the cloud server can work on a finer grain scale. Thus when the user needs quick, less accurate results, then the user can execute it locally on the mobile. On the other hand, if the user wants more accurate, but fairly fast response, the user/system can select the remote execution mode instead.

Right now all the operations that are offloaded within a mobile application is identified manually. In the future, an automatic application partitioning mechanism should be proposed to generate both mobile code and server code. It should be able to analyze the cost relation between different classes in the application and generate the best partition plan based on the user preference and available resources. In addition, it should be able to adjust the execution plan dynamically on the fly.

References

[1] *http://code.google.com/p/jjil/.*

[2] *http://developer.android.com/index.html.*

[3] *http://developer.android.com/reference/android/os/Debug.html.*

[4] *http://www.jhlabs.com/ip/filters/index.html.*

[5] Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop (EW'10)* (New York, NY, 2002), ACM, pp. 87–92.

[6] Chun, B., and Maniatis, P. Augmented smartphone applications through clone cloud execution. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2009).

[7] Flinn, J., Park, S., and Satyanarayanan, M. Balancing performace, energy, and quality in pervasive computing. *Proceedings of the 22nd International Conference on Disributed Computing Systems* (2002).

[8] Giurgiu, I., Riva, O., Juric, D., Krivulev, I., and Alonso, G. Calling the cloud: enabling mobile phones as interfaces to cloud applications. In *Middleware 09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (2009).

[9] Gu, X., Messer, A., Greenberg, I., Milojicic, D., and Nahrstedt, K. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing 3* (2004), 66–73.

[10] Kim, S., Rim, H., and Han, H. Distributed execution for resource-constrained mobile consumer devices. *IEEE Transactions on Consumer Electronics* (2009).

[11] Kumar, K., and Lu, Y.-H. Cloud computing for mobile users: Can offloading computation save energy? *Computer 43* (2010).

[12] Lakshman, Y., and Young, C. Protium, an infrastructure for partitioned applications. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2001).

[13] Li, Z., Wang, C., and Xu, R. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES'01)* (New York, NY, USA, 2001), ACM, pp. 238–246.

[14] Miettinen, A. P., and Nurminen, J. K. Energy efficiency of mobile clients in cloud computing. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010).

[15] Newton, R., Toledo, S., Girod, L., Balakrishnan, H., and Madden, S. Wishbone: Profile-based partitioning for sensornet applications. In *Proceeding of USENIX Symposium on Networked Systems Design and Implementation* (2009).

[16] Satyanarayanan, M., Bah, P., Caceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *Journal of IEEE Pervasive Computing 8* (2009).

[17] Su, Y., and Flinn, J. Slingshot: deploying stateful services in wireless hotspots. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), ACM, pp. 79–92.

[18] Wang, C., and Li, Z. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)* (New York, NY, 2004), ACM, pp. 119–130.