Assessing the Benefits of Computational Offloading in Mobile-Cloud Applications

Tahmid Nabi Pranjal Mittal
Pooria Azimi Danny Dig
Oregon State University, USA
nabim,mittalp,azimip,digd@oregonstate.edu

Eli Tilevich
Virginia Tech, USA
tilevich@cs.vt.edu

Abstract

This paper presents the results of a formative study conducted to determine the effects of computation offloading in mobile applications by comparing *application performance* (chiefly energy consumption and response time). The study examined two general execution scenarios: (1) computation is performed locally on a mobile device, and (2) computation is offloaded entirely to the cloud. The study also carefully considered the underlying network characteristics as an important factor affecting the performance. More specifically, we refactored two mobile applications to offload their computationally intensive functionality to execute in the cloud. We then profiled these applications under different network conditions, and carefully measured *application performance* in each case. The results indicate that on fast networks, offloading is almost always beneficial. However, on slower networks, the offloading cost-benefit analysis is not as clear cut. The characteristics of the data transferred between the mobile device and the cloud may be a deciding factor in determining whether computation offloading would improve performance.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Complexity Measures, Performance Measures

Keywords Cloud offloading, Mobile applications, Energy efficiency, Execution performance

1. Introduction

Mobile application developers often need to increase a mobile application's performance. The advancement of cloud and distributed systems as well as growing bandwidth, can greatly enrich the capabilities of today's pervasive mobile devices. Cloud computing provisions resources, including processor, memory, and storage, not physically present on a local device. Unlike traditional mobile applications, a mobile-cloud application extends its execution beyond a local device, utilizing the cloud to perform some or all of time-consuming computations or queries which are beyond the device's capacity.

However, application developers lack clear guidelines to determine whether offloading computation to the cloud would improve performance. It may not be advantageous to offload every computation in every situation. When determining whether a computation is to be offloaded, one must balance trade-offs, affected by several factors. In the past, these factors have been studied in depth for traditional mobile applications, but there is a need for a comprehensive study specifically for cloud-backed mobile applications, also known as *distributed mobile applications*. This paper reports on a formative study we conducted to answer the following research questions:

- **RQ1:** What factors *(e.g., input size, bandwidth)* improve *application performance* when offloading the computation?
- **RQ2:** Which of these factors are application-specific, and which are device-specific?
- **RQ3:** How do variations in these factors affect application performance for local and offloaded computations, and can these factors alone inform whether one should offload a computation at runtime?

To conduct our study, we identified a representative domain that might benefit from offloading; Optical Character Recognition (OCR). Our reasons are outlined in section 3.1. We then refactored two OCR applications (1 for Android and 1 for iOS) to offload their computation to the cloud. Using 4 different smartphones, we measured these *app performance* metrics in different network situations (outlined in section 2.3.5 and table 2). We used multiple profiling tools, and profiled our apps several times. We used 10 representative

test images, ranging in dimensions, file size, and complexity, investigating the effects of these factors on application performance.

The results show that in almost all cases, offloading seemed beneficial, but only on fast networks (WiFi or LTE). On slower networks, the offloading benefits depended on the "complexity" of the image. Complexity is related to but also different from image size or even the number of characters in an image; some images are inherently more "complex" to OCR. We expand upon this in the related work section.

This paper makes the following contributions: We identify use cases when computation offloading might be beneficial, identify some quantifiable "app performance" metrics, provide guidance on refactoring techniques on different mobile platforms (e.g., tools and challenges) with a focus on profiling tools, provide detailed data (about 200 data points: 4 smartphones, 10 images, 3–8 network conditions), and finally, provide some hypotheses and directions for future research.

2. Methodology and Tools

Network bandwidth, CPU utilization, and memory usage can be measured directly from the device or in the development environment (emulator/simulator). Measuring the energy usage of an application is a challenging task. Neither iOS nor Android currently expose the energy usage of each component. To measure energy usage, researchers used special instrumentation. Banerjee et al. [11] measure component utilization (e.g., $LOAD_{WiFi}$ and $LOAD_{CPU}$) using a specialized power meter. Alternatively, software-based solutions like PowerTutor [22] for Android and Apple's Instruments [3] can also be used. We used both these tools in our study.

2.1 Profiled Devices

Table 1 shows the devices we used to do our profiling. We could not use all devices for all network conditions (Edge, 3G, LTE), but we used at least 2 network conditions for each device. Details can be seen in Table 5. On the whole, we profiled 17 network conditions on 4 mobile devices.

Table 1. Profiled Devices

	CPU	Memory (MB)	Battery Capacity (mAh)
Server	Intel Xeon 2.4GHz (E5-2630L v2)	512	–
iPhone 6	Dual-core 1.4 GHz Cyclone (Apple A8)	1024	2,915
iPad 4	Dual-core 1.4 GHz (Apple A6X)	1024	11,560
Moto X	Dual-core 1.7 GHz Krait (Qualcomm Snapdragon)	2048	2,200
Walton H2	Quad-core 1.2 GHz Cortex A7	1024	2,050

2.2 Android Profiling

Profiling Android applications is a challenging task because a single tool that can profile every performance related aspect of an application like energy usage, response time, memory usage, and network usage is not available. As a result, in our study we used a variety of tools to profile different performance metrics and assimilate data in the end.

2.2.1 Energy Profiling

For energy profiling we use the PowerTutor application. Zhang et al. [22] present PowerTutor's computation model. PowerTutor profiles the energy comnsumption of four components: LCD, CPU, WiFi and 3G antenna.

The developers of PowerTutor used the following steps to generate their Power Model:

1. Obtain the battery discharge curve for each individual component on a particular device using built-in battery voltage sensors.
2. Determine the power consumption for each component state.
3. Perform regression to derive the power model.

2.2.2 Network Profiling

The Android SDK ships with a debugging tool called the Dalvik Debug Monitor Server (DDMS). DDMS communicates with a client device via a tool called adb (android debug bridge).

When DDMS is started, it starts the adb server if it was not already running. The adb server then sets up connections to emulator/device instances. A VM monitoring service is setup, and DDMS uses adb to connect to the VM's debugger and retrieve information. DDMS is a powerful tool and allows us to profile several aspects of an Android application.

DDMS includes a Detailed Network Usage tab that can track when an application is making network requests. Using this tool, we monitored the frequency of data transfers, and the amount of data transferred during each connection.

2.2.3 Memory and CPU Profiling

Using DDMS, we invoke garbage collection, which collects unused heap data. When this operation completes, we inspect the object types and the memory that has been allocated for each type. We also inspect the heap size, the allocated memory, and the free memory.

Using the shell provided by adb, we invoked the unix command "top" and obtain the cpu percentage used by a particular process. This enabled us to profile the CPU usage of an application.

2.2.4 Measuring Response Time

We interpret response time as the time it takes for the application to receive an input for subsequent computation and then return the result of that computation to the user. It is

important to isolate the computation we offloaded from the rest of the UI-related computations.

Based on this definition, we add our instrumentation code to the investigated applications to measure and log response time. We later use *logviewer* in DDMS to obtain the measured response time.

2.3 iOS Profiling

For iOS application profiling, we use the Instruments [3] tool bundled with Apple's IDE, Xcode [10]. Instruments is "a performance-analysis and testing tool for dynamically tracing and profiling OS X and iOS code" [3]. Instruments profiles different aspects of applications running inside iOS Simulator, as well as applications running on the physical devices. In either case, it uses DTrace [2] for low-overhead sampling.

Instruments can profile many aspects of an iOS application's behavior such as: energy usage level, CPU activity (per-process and per-thread), Memory allocation, release, and possible leaks, File and socket access, Networks activity, OpenGL activity, User interface events.

Instruments profiles everything at the same time, which, unlike Android, allows us to easily combine them into a single report.

The unified nature of these profiling tools allows for better understanding of different aspects that can affect energy consumption.

2.3.1 Energy Profiling

Instruments traces the energy usage of an application in a scale of 0 to 20, with 0 being the lowest energy draw. At present, it does not offer fine-grained results as to which component (e.g., LCD, CPU, various antennas) consumes more energy, and the reported energy impact is simply an aggregate of all components. The reported energy usage level has an inverse linear relationship with the time that app can be run continuously until the battery is completely depleted (20 hours minus energy usage level). An app with energy usage of 4 runs out the battery (of the device reporting the measure) in 16 hours. We use this formula to convert the scale into the number of "minutes" of available battery each computation has consumed.

We run our test applications on the device in "release mode", to ensure realistic measurements.

Energy reporting is accurate only when the device is not being charged; but we need the device to be connected to a computer for actual profiling. So we use a USB hub that doesn't draw as much current as the device as an intermediary, so, even though the device is connected to the computer, it completely runs off its own battery.

2.3.2 Network Profiling

Instruments (and Xcode) provide detailed, exhaustive information about an application's network activity, including the number of bytes and packets sent/received, the type and protocol of each packet, the number and rate of dropped packets, and other statistics.

Table 2. Network Conditions

Network Condition	In/Out Bandwidth (MB/s)	In/Out Delay (ms)	Packet Loss (%)
WiFi - High Speed	60 / 60	0 / 0	0%
WiFi - ADSL	5 / 1	20 / 20	0%
LTE	10 / 3	45 / 45	0%
Excellent 3G	1 / 0.78	100 / 100	0%
Edge	0.64 / 0.32	300 / 300	0%
Lossy 3G	1 / 0.78	500 / 500	10%

2.3.3 Memory and CPU Profiling

Instruments also measures the physical and virtual memory allocated to each process:

CPU utilization is also provided on a per-process, per-thread, and even per-queue basis, allowing for even more fine grained analysis.

2.3.4 Measuring Response Time

As with the Android application, we plan to change the source code of both the original and the refactored applications to measure exactly how much time the computation takes (including the time spent sending data to and receiving from the server). This allows us to both form a baseline for comparison not only between the original and refactored applications on the same platform, but also between the two platforms.

2.3.5 Network Link Conditioner

One of the most important factors in computation offloading is the network used to transfer data between the local device and the Web service. The energy used by different antennas (WiFi, Edge, 3G, LTE, Bluetooth) depends not only on the type of the connection, but also on the signal reception and strength [14].

Low signal reception drastically increases latency and energy consumption (P_{tr}) while decreasing the available bandwidth (B). It means that in such situations, only the most computationally expensive operations should be offloaded.

Network Link Conditioner is a tool provided by Apple that can simulate various network conditions, including reduced bandwidth, high latency, DNS delays, and packet loss. It allows fine grained control and also ships with presets for simulating WiFi, 3G, DSL, Edge, High Latency DNS, and Lossy Networks. We use these different conditions when profiling the energy and network usage of our refactored application. Table 2 displays the specifics of each condition.

3. Our Approach

3.1 OCR: A Representative Use Case

We would classify OCR (Optical Character Recognition) as a moderately computationally intensive problem. We want the app (computation involved) to fall in the region where the benefit due to computational offloading could switch due to small changes in realistic values of independent variables like input size, mobile processing power, bandwidth, etc. This apart from helping us do a comparative study could help us demonstrate that dynamic computational offloading decisions might be possible for certain types of applications or computations. By "realistic values" we imply those values that are closer to what could be encountered in real life as on date. With growth in computational ability, bandwidth available, what counts as a close to real-life value for an independent variable might change.

There are quite a few OCR engines available, and some are open-source. The performance and accuracy of such engines varies. In our current work we are focusing on using the tesseract engine[6] and open-source Android apps that make use of the tesseract engine to do OCR. Tesseract [19] is one of the most accurate OCR engines freely available. It was one of the top 3 engines in the 1995 UNLV Accuracy test and has continually grown to become more accurate and efficient.

3.2 Test Images

We used a total of 10 representative images that vary in file size, dimensions, number of characters, and quality of scan (table 3). We specifically chose such a diverse set of images to investigate effects of each of these parameters. The images can be viewed online [8]. They are showcased in Figure 1 (scaled to fit on this paper).

3.3 Refactoring

We refactored two applications, "Simple Android OCR" [4] and "Tesseract-OCR-iOS" [7], to make use of the cloud. These applications used the Tesseract library and ran entirely locally. We wrote a Web service, "ocrbackend" [9], that uses the same library to perform the OCR computation completely on the cloud and deployed it on 1 core virtual machine on Digital Ocean [1]. After refactoring, our applications can optionally use this Web service to offload the computation.

Using a Web service that uses the same OCR library enables us to isolate the effects of the offloading more precisely, as the algorithm and the parameters used are the same both on the device and on the server.

4. Results

4.1 Android

We tested each image on two devices: a Moto X and a Walton H2. We did not use any simulated network conditions for either of these two devices. Instead, for the Moto X we profiled for WiFi and 4G LTE networks, and for Walton H2 we profiled for WiFi and Edge networks.

To ensure reliability, we ran each test thrice and averaged them across the three tests. We ran the experiments a total of 90 times (two devices, 5 images, 3 network conditions for each device, three tests each).

Before starting the test, the device was rebooted. During the test it was ensured only the profiled app was running. Before each measurement, we killed and restarted the app. Before taking measurements, we waited until the CPU activity became zero and energy consumption by the application had stabilized after startup.

On Android, for WiFi, for both our devices, for image of any size and complexity, response time and CPU utilization improved when computation was offloaded to the cloud. Memory usage was typically constant across both local and WiFi. However, for very large and long image the local computation consumed more memory than other conditions. For energy consumption, no conclusion could be drawn for shorter images because energy consumption decreased for one device and increased for the other when using WiFi. However, for images with a high amount of text content $image7 and image7\_2$, energy consumption improved when computation was offloaded to cloud.

For 4G LTE network condition, for image of any size and complexity, response time and CPU utilization improved when computation was offloaded to cloud. Memory usage was typically constant for both local and 4G LTE network, except that local computation consumed more memory for very large image with long content. If computation was offloaded to cloud, energy consumption typically increased. However, for very large image with high content of text, offloading to cloud was more energy efficient.

For Edge network condition, response time and energy consumption worsened when computation was offloaded to cloud while memory and cpu usage remained constant.

To summarize, we found that **response time and CPU utilization always improves if computation can be offloaded to cloud under fast network conditions (WiFi, LTE)**. For small images, it was not conclusive whether offloading to cloud even under fast network conditions was energy efficient. However, **offloading computation of images with high amount of text content was observed to be energy efficient under fast network conditions (WiFi, LTE)**. **For poor network conditions (Edge), offloading computations ended up worsening both response time and energy efficiency** and left other metrics unchanged.[1]

4.2 iOS

We tested each image on two iOS devices: a WiFi-only 4th gen. iPad, and an iPhone 6. Because iPad could not connect to cell networks, we used Network Link Conditioner to "simulate" those conditions, i.e., restrict bandwidth and

[1] All our experimental data has been made available online [5].

Table 3. Test Images

image_#	Name (size, quality, length)	Width x Height (px x px)	Size (KB)	Text Length (lines)	Text Length (characters)
0	small, crisp, very short	400 x 216	12	2	10
1	small, handwriting scan, short	314 x 300	13	7	37
2	medium, good scan, medium	884 x 809	126	37	780
3	medium, noisy scan, medium	690 x 667	102	23	1,096
4	big, crisp, short	2048 x 1536	251	6	65
4_2	*small, crisp, short*	*307 x 230*	*20*	*6*	*65*
5	big, sepia scan, long	688 x 1096	101	36	2,069
6	big, bad scan, long	800 x 956	153	122	3,641
7	very big, good scan, very long	3768 x 5256	2,316	72	4,324
7_2	*medium, good scan, very long*	*942 x 1314*	*363*	*72*	*4,324*

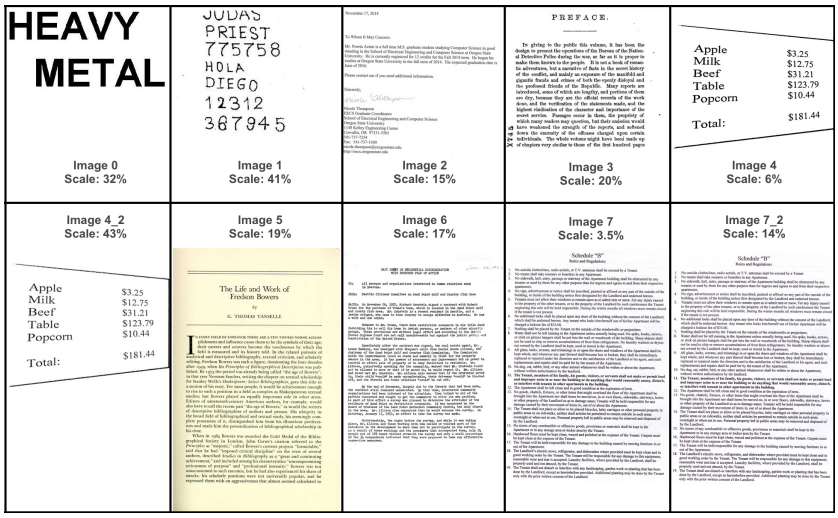

Figure 1. Test Images

introduce delays to simulate e.g., a lossy 3G network, while still using the WiFi antenna. As a result, our energy profiles were not accurate and wee didn't include them in our tables. They were, however, reliable in other "performance metrics" (e.g., response time and memory usage), so we have made them available online [5].

To ensure reliability, we ran each test twice and averaged the results. Overall, the tests were run 280 times (two devices, 10 images, 7 network conditions, two tests each).

Before starting the test, the device was rebooted; during the test, only our profiled application was running; we killed the application and re-loaded it into the device for each measurement; and before starting to profile, we waited until all the assets were loaded, CPU activity dropped to 0%, and energy usage was stable for at least 10 seconds.

Table 4 shows the results of local OCR on the server and iOS devices (for comparison). We found that on iOS, **it is almost always beneficial to offload the OCR computation to the cloud, provided you have a fast WiFi network or are on a LTE network**. In such cases, offloading improves most of our "app performance" metrics: drastically less energy usage, an order of magnitude faster response time, and almost no CPU or memory usage, while network usage naturally increases.

When on a 3G or 2G network, however, the results are not really conclusive—we could not identify a pattern between the image size and whether or not offloading improves "app performance." We suspect that we need to further study another, unexplored factor, called "image complexity."

4.3 Images with Large Font Size

2 of our test images (Images 4 and 7) were unusual in the sense that, when viewed at 100% scale, their font size seemed respectively 150pt and 50pt, as opposed to the usual 18–20pt of other images. We shrunk them significantly (from 2048x1536 pixels to 307x230 pixels in the case of image 4) to study how this affects our "app performance". Tables 6 and 7 show the results (compare to tables 5 and 6 respectively).

Surprisingly, the dramatic decrease in image size did not have a great impact on local response time or energy usage, but it significantly improved the response time and energy usage on offloaded computations. We hypothesize that the reason local computation did not improve is that reducing the size of the image does not change its "complexity" (as defined in the next section), while the improve in speed and battery

Table 4. Local OCR – Time and Energy Usage

		image_0	image_1	image_2	image_3	image_4	image_4_2	image_5	image_6	image_7	image_7_2
Server	**Time (ms)**	38	84	1,372	3,621	359	279	10,504	19,854	7,479	8,651
iPhone 6	**Time (ms)**	440	790	3,700	10,850	1,190	900	29,700	78,480	13,060	19,591
	Energy (min)	0.1 min	0.05 min	0.25 min	0.65 min	0.07 min	0.033	1.62 min	4.87 min	0.63 min	1.27 min
iPad 4	**Time (ms)**	1,180	1,450	8,550	23,780	2,630	1,890	73,110	151,250	28,580	34,750
	Energy (min)	0.13 min	0.15 min	0.63 min	1.53 min	0.12 min	0.05 min	1.55 min	3.08 min	0.8 min	0.44 min

Table 5. Image 0 – Small, crisp, very short and Image 4 – Big, crisp, short

		image_0: small, crisp, very short					image_4: big, crisp, short				
Device	**Network**	**Resp. Time (ms)**	**CPU Used (%)**	**Tx Bytes (B)**	**Mem usage (MB)**	**Energy Usage**	**Resp. Time (ms)**	**CPU Used (%)**	**Tx Bytes (B)**	**Mem usage (MB)**	**Energy Usage**
Server	–	38	97	0	12	–	359	99	0	45	–
iPhone 6	**Local**	440	13	0	29	.1 min	1,190	91	0	41	.07 min
	WiFi - High Speed	300	1	29	1	.07 min	850	2	684	14	.03 min
	WiFi - ADSL	1,050	1	28	2	.08 min	16,230	1	685	13	.18 min
	LTE	620	2	28	2	.25 min	2,470	2	685	13	.27 min
	Excellent 3G	1,680	1	28	2	.28 min	16,090	2	685	14	.38 min
	Edge	5,200	2	28	2	.35 min	22,610	1	683	13	2.25 min
	Lossy 3G	13,610	2	68	3	.53 min	120,690	2	795	14	3.73 min
iPad 4	**Local**	1,180	36	0	39	.13 min	2,830	43	0	41	.12 min
	WiFi – High Speed	280	2	29	2	.07 min	1,020	1	685	14	.05 min
	WiFi – DSL	1,110	3	28	3	.1 min	16,590	2	685	14	.23 min
Moto X	**Local**	1,160	19	0	9	.3 J	2,280	23	0	9	1.87 J
	WiFi – High Speed	189	3	12	9	2.7 J	1,623	4	266	9	2.83 J
	LTE	626	3	12	9	5.37 J	1,612	4	264	9	4.8 J
Walton H2	**Local**	1,248	11	0	4	.6 J	2,447	22	0	5	.9 J
	WiFi - High Speed	255	4	12	4	.24 J	1,401	5	266	5	.6 J
	Edge	6,004	3	12	4	1.5 J	74,316	2	264	5	1.65 J

Table 6. Image 4.2 – Small, crisp, short and Image 7 – Very large, good scan, very long

		image_4_2: small, crisp, short					image_7: very big, good scan, very long				
Device	**Network**	**Resp. Time (ms)**	**CPU Used (%)**	**Tx Bytes (B)**	**Mem usage (MB)**	**Energy Usage**	**Resp. Time (ms)**	**CPU Used (%)**	**Tx Bytes (B)**	**Mem usage (MB)**	**Energy Usage**
Server	–	359	97	0	12	–	7,479	99	0	123	–
iPhone 6	**Local**	900	31	0	37	.03 min	13,060	94	0	133	.63 min
	WiFi - High Speed	600	3	30	3	.05 min	10,610	4	1830	88	.23 min
	WiFi - ADSL	790	4	30	3	.1 min	58,210	3	1830	83	.58 min
	LTE	1,220	2	31	3	.2 min	20,630	5	1830	83	.52 min
	Excellent 3G	1,120	3	30	3	.58 min	43,830	4	1831	85	1.25 min
	Edge	1,180	4	30	3	.88 min	119,320	5	1830	88	2.75 min
	Lossy 3G	1,210	5	44	3	1.08 min	264,130	6	2170	86	7.35 min
iPad 4	**Local**	1,890	45	0	45	.05 min	28,580	85	0	220	.8 min
	WiFi – High Speed	1,640	2	30	6	.08 min	12,260	6	1832	83	.28 min
	WiFi – DSL	2,380	3	30	6	.08 min	60,090	3	1830	86	.53 min
Moto X	**Local**	1,133	4	0	17	.97 J	28,359	47	0	14	21.6 J
	WiFi – High Speed	643	3	22579	17	1.3 J	13,065	25	2463	12	5.5 J
	LTE	708	4	21063	17	3.47 J	11,284	25	2461	17	9.47 J
Walton H2	**Local**	1,334	5	0	4	.8 J	40,236	24	0	18	13.27 J
	WiFi - High Speed	476	3	21290	4	.5 J	12,778	9	2469	13	3.97 J
	Edge	10,171	2	25468	4	.7 J	706,344	13	2472	12	4.3 J

Table 7. Image 7.2 – Medium, good scan, very long

Device	**Network**	**Response Time (ms)**	**Average CPU Utilization (%)**	**Transmitted Bytes (B)**	**Memory Usage (MB)**	**Total Energy Usage**
Server	–	8,930	100	0	26	–
iPhone 6	**Local**	19,590	88	0	54	1.27 min
	WiFi - High Speed	9,240	3	1385	8	0.08 min
	WiFi - ADSL	33,130	3	1388	11	0.78 min
	LTE	12,960	4	1385	7	0.28 min
	Excellent 3G	43,010	5	1385	8	0.3 min
	Edge	67,130	5	1390	9	0.92 min
	Lossy 3G	93,230	6	1520	6	1.58 min
iPad 4	**Local**	34,570	84	0	67	0.43 min
	WiFi – High Speed	9,460	3	1386	13	0.15 min
	WiFi – DSL	41,780	5	1386	14	0.23 min
Moto X	**Local**	1,296	4	0	17	1.6 J
	WiFi – High Speed	172	3	13842	17	1.23 J
	LTE	230	3	13738	17	3.03 J
Walton H2	**Local**	1,452	13	0	4	1.6 J
	WiFi - High Speed	119	3	13290	4	0.85 J
	Edge	5,635	2	13721	4	1.65 J

usage on offloaded computations can be entirely attributed to smaller image size and bandwidth usage.

5. Future Work

During our study we discovered that apart from known factors, such as input size, other input characteristics can affect app performance. Two images A & B of the same size can have A taking longer to OCR due to larger number of characters, using a font that is harder to read, or other factors that are briefly explored by Smith [19]. We collectively refer to such characteristics as "complexity" which is hard to measure quantitatively. Complexity in the context of OCR might be different from complexity in image segmentation.

In this paper, we studied the effects of variation in application-specific and device-specific factors for OCR computations (for both local and offloaded computations). Much of the findings can be generalized, as OCR is nothing but a general computation task which could be replaced by some other task and still lead to similar results when the factors like input size are varied. However, certain application-specific factors (like "complexity of image") seem to be dependent on the type of computation. There is scope for doing the same study in context of more mobile applications because of computation dependent factors like "input complexity" might vary from study to study. "Input complexity" is also an ambiguous term and needs to be defined properly.

6. Related Work

Barbera et al. [12] investigate the bandwidth and energy costs of mobile cloud computing. It models the use of the cloud by associating each real device with its corresponding cloud-clone. Clones are categorized as offloading either computation or storage. The energy cost and bandwidth needed for both types are investigated. The paper finds that clones offloading computation use more energy and bandwidth for synchronization compared to clones offloading storage. Compared to our study, that investigation was carried out from the perspective of the mobile device itself rather that of the underlying application. This study did not look at type of computation/functionality offloaded to the cloud.

Kumar et al. [14] present a mathematical model to compare the energy usage in local vs. offloaded computations. The limitation of the paper is that it only provides a model for computing energy usage, which is just one of the factor to judge "app performance". Net computation time, memory and data consumption are the other factors. There is a tradeoff between these factors. Example: Net computation time can be reduced if we have a higher processing power which would in turn consume more energy.

In another part, Kumar et.al [13] provide a mathematical model for deciding when offloading computation improves performance. The model takes into account the size of computation, processing power of mobile device, processing power of server, the size of data to be transmitted over the network and network bandwidth. This paper also presents a survey of previous research work done on computation offloading.

Zhang et al. [23] automatically refactor Java code for offloading. Remotely executable classes are identified and then repackaged as a .jar and run on a remote server in the cloud. The main limitation of their approach lies in their movable class identification policy. They identify immovable classes as those which either extend/implement/uses Android system classes or those which have the "native" keyword in method names. Thus, their strategy is heavily coupled with the Android OS and must evolve alongside it. Also, whether offloading the classes is actually beneficial or not is not explored.

Kwon and Tilevich [15] present an approach that improves the energy efficiency of mobile applications by enhancing the code with dynamic adaptation capabilities. The decision of whether to offload to the cloud and which part of the application's functionality is determined at runtime taking into account both the mobile device at hand and the current execution environment.

Ray Smith [19] describes the architecture of the Tesseract OCR engine, used as the representative case in our study. The paper describes in detail how Tesseract finds and recognizes letters, words, and lines; how it classifies these findings; and how the training corpus is used to build the classifier.

Nikzad et al. [18] argue that writing energy efficient code is difficult and obscures business logic code. They develop a middleware and an annotation-framework on top of it called *Annotated Programming for Energy Efficiency*. Using this framework, developers annotate code they want to make energy-efficient, which are then preprocessed by the middleware to generate actual energy efficient code. Compared to us, they consider only energy efficiency and do not offload any computation to the cloud.

Xu et al. [20] through low level optimization techniques propose improvements to the energy efficiency of the Gmail Android app, which continuously interacts with the cloud. The investigation finds that Gmail's major energy consumption occurred when it used 3G connection for syncing with the cloud. They wrote firmware to optimize usage of 3G connection. Their finding of 3G connection consuming greater energy matches the observations from our experiments as well.

Mtibaa et al. [17] present app performance benefits in the same way as we do, i.e., primarily in terms of net computation time and energy consumption. Other than that their work was not much related to our study. They study effects of mobile computation offloading to a cloud of mobile like or low capacity devices which they refer to as a "Mobile Device Cloud".

There is some work done in the field of image complexity and spatial information by Yu et al. [21]. It is useful for qualitatively representing image complexity in terms of ease of compression. This is in coherence with our hypothesis that

input complexity must relate to the algorithm or computation to be performed as we suggested in the future work section. We did not find existing research work in relation to image complexity in context of OCR.

7. Conclusions

This research explored which factors improved app performance when computation is offloaded to the cloud, the nature of these factors, and how variations in these affect app performance for local and offloaded computations. OCR is an ideal use case for investigating these research questions. We refactored OCR apps on iOS and Android to offload the OCR computations to the cloud, and investigated app performance across a variety of inputs and network conditions. We found that under fast network conditions, app performance always improved when computation was offloaded. We also found that compressing images improved app performance for offloaded computations.

Acknowledgments

We thank Harshit Gupta of IIT BHU, Varanasi for providing support with refactoring of Android applications.

This research is supported in part by the National Science Foundation through Grants CCF-1116565 and CCF-1439957.

References

[1] Digital Ocean. `https://www.digitalocean.com/`. Accessed: 2014-12-10.

[2] DTrace. `https://wiki.freebsd.org/DTrace`. Accessed: 2014-11-11.

[3] Instruments. `https://developer.apple.com`. Accessed: 2014-11-11.

[4] Simple Android OCR application. `https://github.com/GautamGupta/Simple-Android-OCR`. Accessed: 2014-11-11.

[5] Study's raw data. `https://www.dropbox.com/sh/a6nt58u2mutwelc/AACF7eSHsnbMCYCzOj8gH1Mia?dl=0`.

[6] Tesseract OCR engine. `https://code.google.com/p/tesseract-ocr`. Accessed: 2014-11-11.

[7] Tesseract OCR iOS application. `https://github.com/gali8/Tesseract-OCR-iOS`. Accessed: 2014-11-22.

[8] Test images. `https://www.dropbox.com/sh/knhtx8llquww5lr/AADh58CNfzNmzSFaQktpXI4Fa?dl=0`.

[9] Web OCR backend. `https://github.com/pramttl/ocrbackend`.

[10] Xcode. `https://developer.apple.com/xcode`. Accessed: 2014-11-11.

[11] A. Banerjee, L. Kee, C. Sudipta, and C. Abhik. Detecting Energy Bugs and Hotspots in Mobile Apps. 3, 2012.

[12] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *INFOCOM, 2013 Proceedings IEEE*, pages 1285–1293. IEEE, 2013.

[13] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[14] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[15] Y.-W. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 170–179. IEEE, 2013.

[16] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[17] A. Mtibaa, K. Harras, and A. Fahim. Towards computational offloading in mobile device clouds. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 331–338, Dec 2013.

[18] N. Nikzad, O. Chipara, and W. G. Griswold. Ape: An annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 515–526, New York, NY, USA, 2014. ACM.

[19] R. Smith. An overview of the tesseract ocr engine. In *ICDAR*, volume 7, pages 629–633, 2007.

[20] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 55–68, New York, NY, USA, 2013. ACM.

[21] H. Yu and S. Winkler. Image complexity and spatial information. In *Quality of Multimedia Experience (QoMEX), 2013 Fifth International Workshop on*, pages 12–17, July 2013.

[22] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114, Oct 2010.

[23] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *ACM SIGPLAN Notices*, volume 47, pages 233–248. ACM, 2012.