Power-Aware Wireless Transmission for Computation Offloading in Mobile Cloud

Lei Zhang*, Cong Zhang*, Jiangchuan Liu*, Xiaowen Chu§, Ke Xu‡, Haiyang Wang† Yong Jiang‡
*School of Computing Science, Simon Fraser University, Canada
§Department of Computer Science, Hong Kong Baptist University, Hong Kong
‡Department of Computer Science and Technology, Tsinghua University, China
†Department of Computer Science, University of Minnesota at Duluth, USA
Email: {lza70,congz,jcliu}@cs.sfu.ca, chxw@comp.hkbu.edu.hk,
xuke@tsinghua.edu.cn, haiyang@d.umn.edu, jiangyg@sz.tsinghua.edu.cn

***Abstract*—In today's mobile devices, the battery reservoir remains severely limited in capacity, making power consumption a key concern in the design and implementation of mobile applications. In this paper, we closely examine one widely adopted approach to improve the energy efficiency of mobile applications– adaptively offloading the computation to the remote cloud. In particular, we measure the power consumption of computation offloading for two representative real-world mobile cloud applications under various wireless network conditions and identify the unique features of data transmission for computation offloading. We then formulate the power-aware scheduling problem for computation offloading and present a scheduling algorithm that makes adaptive offloading decisions according to the dynamic network conditions. Simulation results show that our proposed method can achieve better battery performance, which also reveal that computation-intensive and delay-tolerant tasks are more likely to benefit from offloading.**

I. Introduction

Cloud computing and mobile systems perfectly suit each other: mobile platforms need the "infinite" computation ability and the "unlimited" storage from cloud to achieve better availability and reliability; on the other hand, cloud computing is also looking for its killer application to reinforce its dominant position. As an approach to bridge cloud resources and mobile platforms, offloading can alleviate the computation burden of mobile devices at the cost of extra transmission. In order to extend the battery life of mobile devices by offloading, the tradeoff between the power consumption for computation and that for transmission needs to be carefully considered.

Before utilizing cloud resources to assist the execution of certain application, one needs to know how to partition the application into the local part and the cloud part. Normally, decoupling the target application is not a difficult task for desktop PCs in local area networks (LANs) that visit cloud resources through Internet, since the wired connection is fast, stable, persistent, and at low cost. In most cases, a determinate part of computation can be shifted from PCs to the cloud, and the traffic generated during the migration is not a serious concern for LANs.

When it comes to mobile or wireless networks, the situation is totally different, and the problem becomes much more challenging. The key observation is that, mobile devices access Internet via WiFi or cellular networks, and the connection between them experiences spacial or temporal instability. For cellular data networks, the signal strength, which may vary significantly across different areas, has a direct impact on battery power consumption, since both radio power and data rate are affected [1]. While WiFi radios also exhibit significant variation in energy cost per bit at different locations, this is mainly due to the variation in data rate rather than radio power [2]. Even for the WiFi users who keep stationary, the network condition still changes from time to time under different circumstances. For example, if a WiFi network has only one effective connection, the link quality is likely to be good, however, when multiple devices communicate with the access point simultaneously, they may suffer from severe interferences. On the other hand, poor network connection can make computation offloading infeasible, since both of the two goals of offloading (saving battery and improving performance) may fail under certain conditions: higher radio power leads to higher battery power consumption, and lower data rate increases the transmission delay and thereby lengthens the elapsed time for the application's execution. When offloading is no longer applicable or beneficial, computation should be kept locally. Therefore, it is critical to make offloading scheme adaptive to the changing network condition. Unlike offloading from PCs through wired connection, for mobile platforms, the offloading decisions should be made dynamically together with the transmission decisions during the process according to the quality of wireless connection.

To make optimal computation offloading decisions, one need to answer two key questions: what to offload; and when to offload. In this paper, we show that, for mobile systems, these two questions should be considered jointly, as one can have direct impacts on the other in terms of battery consumption and performance due to the nature of wireless channel's instability. To examine the battery power efficiency of mobile computation offloading under various network conditions, we vary the network delay and packet loss rate in our experiments and test the performance with representative real mobile applications. Our measurement shows that unstable wireless connections can affect data transmission significantly during computation

978-1-5090-2279-3/16/$31.00 ©2016 IEEE

offloading, and reduce the battery gain from offloading. As such, both the uplink and downlink traffic need to be carefully scheduled, and, besides deferring the wireless traffic, offloading can be aborted to cancel the transmission. We further present a generic formulation for power-aware transmission in computation offloading, which exploits the unique features different from traditional mobile/wireless data transmission. We propose an effective solution that adaptively schedules the transmission in fine granularity to accommodate the dynamic wireless connection. Trace-driven simulation results prove the superiority of our proposed method, which also show that computation-intensive and delay-tolerant workloads are preferable for mobile computation offloading.

II. Related Work

With recent advance in cloud computing, computation offloading has received great attention. The attractive features of cloud computing such as high availability, high reliability, and "infinite" resources have made offloading a very promising technique for mobile platforms. Systems that offload computation to Virtual Machine (VM) instances or clouds have been introduced by researchers. MAUI [3] was proposed to offload methods from .NET applications to a remote runtime environment based on a history of power consumption, which has considered some low-level challenges such as failure handling and program state transferring. One drawback of MAUI is that it required programmers to annotate methods that can be offloaded for remote execution. Similar to MAUI, CloneCloud [4] partitioned applications using a framework that combined static program analysis with dynamic program profiling and optimized execution time or power consumption using an optimization solver. CloneCloud focused on a detailed design for state migration and merging, and allowed executing virtualized methods remotely to call native functions.

To provide secure computation offloading, Enterprise Centric Offloading System (ECOS) was proposed [5], in which data/state was categorized into different privacy levels, and private data was transferred through the TLS-encrypted connection. Using smartphone VM image inside the cloud for handling computation offloading, ThinkAir [6] targeted a commercial cloud scenario with multiple mobile users instead of computation offloading of a single user, which focused not only on offloading efficiency and convenience for developers, but also on the elasticity and scalability of the cloud for the dynamic demands of variant customers. Different from the previous studies that have focused on service partitioning for computation offloading and the corresponding low-level details such as program profiling and state migration, we consider optimal schedule for the offloading request and the required transmission based on the application demands and the network condition.

Managing battery power consumption in network activities is a critical issue for mobile devices. Numerous papers have been presented to improve the energy efficiency in wireless transmission. Ra et al. [7] discussed the trade-off between QoS and delay of data transmission for mobile platforms and presented a stable and adaptive link selection algorithm (SALSA). SALSA is an optimal online algorithm for energy-delay trade-off based on the Lyapunov optimization framework to decide whether and when to defer a transmission until a less power-consuming WiFi connection becomes available. Another power-aware transmission scheme, Catnap [8], exploited the bottlenecks of wireless and wire links and utilized an application proxy to decouple data units into segments. With certain modifications on the network gateway (e.g. AP in WiFi, BS in cellular network), it scheduled the segments to be transferred in a burst and merged tiny gaps between packet transmissions into meaningful sleep intervals.

On the practical side, Bartendr [1] demonstrated that strong signal reduces energy cost from empirical measurement. They then developed power-aware scheduling algorithms for different workloads (background synchronization traffic and video stream traffic) based on the signal prediction by location and history. Shu et al. proposed eTime [9], a novel energy-efficient data transmission strategy between cloud and mobile devices, which selected the network interface of better connectivity between 3G and WiFi and chose the proper timing to transfer data. The proposed algorithm has been implemented and applied to a variety of real-world mobile social applications in their following work [10]. Another recent work has studied closely the power models of the WLAN, Third Generation (3G), and Fourth Generation (4G) interfaces of smartphones, specifically for task offloading [11]. Although these works have reduced power consumption for data communication on mobile devices, few of them have investigated the distinct characteristics of data transmission in computation offloading, which will be further explained in the next section.

III. Measurement

In this section we conduct a measurement study to achieve the following goals:

- Compare the power consumption of local execution and offloaded execution for the target applications, and verify the potential benefit from computation offloading.
- Identify the variation in performance and energy efficiency of computation offloading under different network conditions.

A. Devices, Tools and Configurations

Our experiments were performed on a 32 GB Google Nexus 7 tablet that can access Internet through WiFi and mobile networks. The test device runs the Andriod 4.2.2 OS, and has a battery of 4325 mAh at 3.7 volts. In order to measure the actual power consumption, we used a digital multimeter to record the current transferred between the battery and the tablet since the supply voltage can remain stable in a long period of time. The current can be sampled once per second with the accuracy of 10 mA, and be recorded by a software. We unplugged the battery from the device, and wired them up with the digital multimeter as a serial connection in the circuit. Besides recording the current consumption, we also collected the execution traces of our test applications. We

developed a software profiler to profile the CPU usage and the wireless traffic on the mobile device. Another analysis tool, Application Resource Optimizer (ARO) [12], was also adopted in our experiments, which can capture the packets and analyze the radio states during the execution. The two profilers were running as the background processes when our testing mobile applications executed.

Since one of our goals is to investigate the power impact of dynamic wireless networks during offloading, we emulated different network conditions. In our experiments, a desktop PC that runs the Linux OS was used to bridge the gateway to Internet and the wireless AP that the test tablet was connected to. To vary the network properties, we used `netem` [1] on the desktop PC, which provides the network emulation functionality and is available in current Linux systems. Its features include wide area network delays with different delay distribution, packet loss, packet duplication, packet corruption and packet re-ordering. In order to ensure that the variation of battery power consumption is caused by the changing network conditions, the interferences from other factors should be minimized. First, we disabled the CPU frequency auto-scaling on the testing mobile device, and kept the CPU working at a fixed frequency. Second, as the screen usually takes up most of the consumed battery power on a mobile device, the auto adjustment of screen brightness was also disabled. Finally, we killed all the irrelevant and unnecessary processes before profiling the execution of our test applications.

B. Test Applications

Two realworld mobile applications were selected as the test applications to conduct the experiments. The first application is an open source chess game, DroidFish [13], which has an integrated chess engine in the program and is also able to configure a network chess engine. A chess engine is a computer program that analyzes chess positions and makes decisions on the best moves. Although the chess engine decides what moves to make, it typically does not interact directly with the user, instead, it communicates with the graphical user interface (GUI) via the Chess Engine Communication Protocol. The chess game has three game modes: player vs player, player vs computer, and computer vs computer. To test this application, we set the chess game into computer vs computer mode, in which no human input is required, and the program only communicates with the chess engine. In this mode, a significant amount of computation needs to be done by the chess engine, and only a little communication is required since both the chess positions and the chess moves can be expressed as plain text. Therefore, the chess game is promising for battery saving from computation offloading as it is very computation-intensive and requires little data transmission. The possibility of using different chess engines in the application allow us to compare the local execution with the offloaded execution: when the chess game works with a local chess engine, the computation are executed locally on the mobile device, while the computation can be offloaded remotely if a network chess engine is configured. In our experiments, the network chess engine was configured on a laptop with Core i3-2310M CPU and 4 GB RAM which was connected to Internet through a wireless AP.

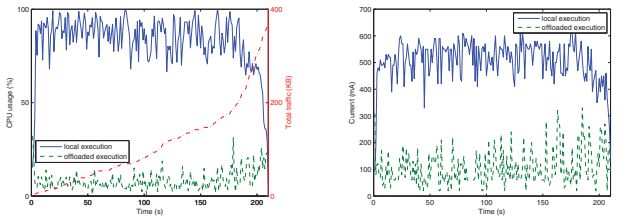
We chose OnLive, one of the pioneering commercial cloud gaming platforms, as the other test application. Advances in cloud technology have expanded to allow offloading not only of traditional computations but also of such more complex tasks as high definition 3D rendering, which turns the idea of cloud gaming into a reality. In a cloud gaming system, a player's commands must be sent over the Internet from the its thin client to the cloud gaming platform. Once the commands reach the cloud gaming platform they are next converted into appropriate in-game actions, which are interpreted by the game logic into changes in the game world. The game world changes are then processed by the cloud system graphical processing unit (GPU) into a rendered scene. The rendered scene is usually compressed by the video encoder, and then sent to a video streaming module, which delivers the video stream back to the thin client. Finally, the thin client displays the video frames to the player. To test OnLive, we ran a cross-platform game, Osmos [14], through an OnLive mobile client as well as locally on our test device, as the game is also available as a mobile app in Google Play Store.

C. Local Execution vs Offloaded Execution

To prove that computation offloading can bring potential battery savings on mobile devices, we next compare the local execution and the offloaded execution of the two test applications. For the chess game, we profiled the current consumption and the execution trace when the program used the local chess engine and the network chess engine respectively. For the second application, we first ran the mobile game as an independent app on the test tablet. As its comparison, we then played the same game integrated in OnLive. As we cannot guarantee that every time we have exactly the same inputs in the game, in order to eliminate the inferences from human interactions, we did not generate any user input when we recorded the current consumption and the execution traces during the video playback.

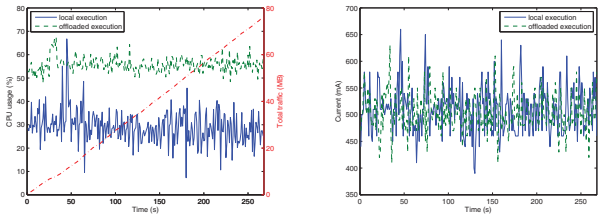
Figure 1(a) shows the CPU usage and the total traffic during one execution of the chess game, which provides the strong evidence that, for computation-intensive tasks, computation offloading can mitigate the burden on CPUs at the low cost of extra data transmission. From Figure 1(b) we can see that it significantly reduces the power consumption if the test device offloads the heavy calculation to the cloud. It should be noted that, at the end of the chess game, the data transfer rate of the offloaded execution increases, and the CPU usage of the local execution decreases. The reason is that, there are usually only few pieces left on the board when the chess game comes to the end, and it needs less computation to decide the best move at that time, and thus the network engine responds more quickly and the communication becomes more frequent.

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem



(a) Execution trace (b) Current consumption

Fig. 1: Comparison between local execution and offloaded execution for the chess game



(a) Execution trace (b) Current consumption

Fig. 2: Comparison between local execution and offloaded execution for OnLive

The previous results prove that computation offloading has the potential to save mobile device's battery for computation-intensive applications such as the chess game. However, the execution traces for the mobile game and OnLive show different results. In Figure 2, both the CPU usage and the current consumption are higher when the mobile game is executed in the OnLive platform. This can be explained by the fact that the two programs have totally different implementations. When the mobile game running as a installed app, it needs more GPU computations to render the game scenes, while when it runs in OnLive it requires more CPU computations to decode the received gaming video. Moreover, the goal of OnLive is to provide ubiquitous gaming experience that normally cannot get from mobile devices rather than saving their batteries. And the test game itself does not belong to the kind of games that require high-end hardware and are the main targets of OnLive. Therefore, it is not surprising that OnLive consumes slightly more battery power than the mobile game's local execution.

D. Offloading under Dynamic Wireless Networks

In this section, we investigate how the performance and energy efficiency of computation offloading change when the network condition varies. As mentioned earlier, we used a desktop PC as the network bridge between the test device and Internet in our experiments. We first tested the two applications under the normal and stable wireless connection, and then increased the network latency and the packet loss rate to emulate the dynamic wireless connections.

1) Impact of Delay: Our measurements show that, when the network latency is increased, the power consumption does not vary much for the two test applications. Although the power consumption of computation offloading remains stable, we argue that the increasing delay may hurt the performance of the offloaded tasks. Higher network latency implies that more time is needed to get the response when the application interacts with the cloud. For some applications such as Email, the delay can be masked so that it does no harm to the user experience. For example, in the chess rules, each player has a fixed interval of time to think and make the move; when the network latency increases, the network chess engine has less time to search the best move (less thinking time) as it takes longer to send back the result, but it does not affect the user experience as long as the total delay does not exceed the whole thinking interval. However, for the delay-sensitive applications, the high network latency can result in a significant performance degradation. Studies on traditional online gaming systems have found that different styles of games have different thresholds for maximum tolerable delay [15]. Table I summarizes the maximum delays that an average player can tolerate before the Quality of Experience (QoE) begins to degrade.

Example Game Type	Perspective	Delay Threshold
First Person Shooter (FPS)	First-Person	100 ms
Role Playing Game (RPG)	Third-Person	500 ms
Real Time Strategy (RTS)	Omnipresent	1000 ms

TABLE I: Delay tolerance in online gaming

Added delay	Avg num of frames updated	Represented response time
0	5 frames	0.33 s
50 ms	5.7 frames	0.38 s
100 ms	6.5 frames	0.43 s
150 ms	7.5 frames	0.50 s

TABLE II: Average number of frames updated before the response arrived

To measure the actual impacts that the network delay has on OnLive, we recorded the tablet's screen (the frame rate was approximately 15fps) when we ran the game in OnLive, and analyzed the gaming video frame by frame. Figure 3 shows how the screen updated after one touch was detected on the screen. The left side lists OnLive's responses with the normal connection, while the right side presents the updated frames after we increased the delay by 150ms. With the better wireless connection, it only needs 5 frames to get the response of that action from OnLive, whereas 8 frames are needed for the worst case. As shown in Table II, the average response time increases as the delay gets higher, which implies that the performance of OnLive degrades.

2) Impact of Packet Loss: We next examine the influence of packet loss rate, another important parameter in the wireless testing environment. In our measurements, we found that the power consumption for the two test applications when the packet loss rate varies does not change significantly. However, it is not necessary that the energy efficiency of computation offloading remains unchange. The packet loss can have great negative effects on the quality of network communication, which means that it may cost more battery power to transfer the same amount of data successfully. In other words, the effective throughput on the wireless link drops when the packet loss rate becomes higher.

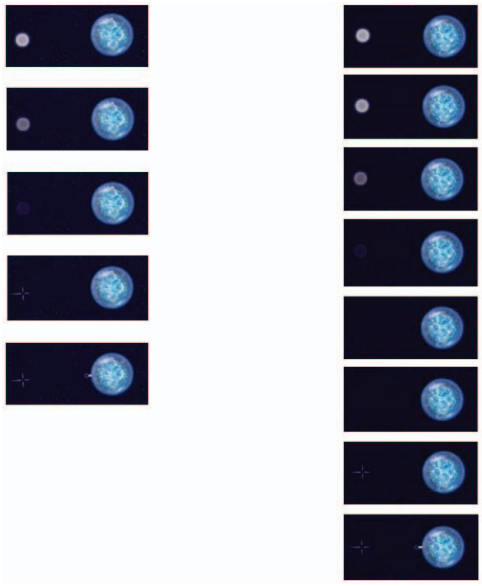


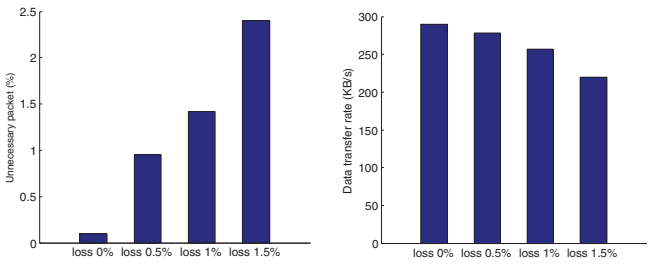
Fig. 3: Captured frames in OnLive with low and high network latency

We captured the transferred packets during the computation offloading for the two test applications. The chess game's GUI communicates with the network chess engine through a TCP connection. We classify the packets that should not appear in the normal data transfer as unnecessary packets, such as duplicate ACKs, timeout retransmissions, fast retransmissions (after 3 duplicate ACKs), ACKs of unseen segment and other special packets. For each run of the chess game, we counted the total number of packets and the number of unnecessary packets, and calculated the percentage of unnecessary packets. As shown in Figure 4(a), with the normal stable wireless connection there are only 0.1% unnecessary packets, whereas the abnormal traffic takes up 2.4% of the total traffic if the packet loss rate is increased to 1.5%. Different from the chess game, OnLive streams the gaming videos to clients via UDP. We measured the data transfer rate between the mobile client and the OnLive's cloud sever instance. Figure 4(b) presents the mobile client's average downloading data rate under different packet loss rates, which shows that the downloading rate drops by 24.1% when the wireless connection loses more packets, although the the data rate remains higher than 200KB/s. It should be pointed out that the decreasing data rate not only implies the lower energy efficiency, but it may also hurt the application's performance (in OnLive's case, the image quality of the gaming video was affected).

IV. Problem Formulation and Solution

A. Basic Problem

As the performance and energy efficiency of computation offloading can be largely affected by the various network conditions, based on our observations in the last section, we give a formal description of the general computation offloading scheduling problem in wireless communications for mobile cloud. Our proposed approach focuses on making power-aware offloading decisions and assigning intervals for data transmission.



(a) Percentage of unnecessary packets (b) Transfer rate of OnLive

Fig. 4: Impact of packet loss

Assume that the application has a set of n tasks $J = \{j_1, j_2, ..., j_n\}$, and m time slots $\{t_1, t_2, ..., t_m\}$ in total to execute them. Each task stands for a possible user input, the required computations and the generated results, which can be executed either locally or in the cloud. Taking cloud gaming as an example, in a continuous gaming scene, the rendering of each object/in-game character can be taken as a separate task, since each in-game character can have an independent game logic of responding the gamer's input. We denote a task with a tuple of five elements $j_i = (c_i, t_i^a, t_i^d, d_i^s, d_i^r)$, where c_i is the computation load, t_i^a is the task's start time (e.g., when the user input occurs), t_i^d is the deadline for the task to be done and sent back, d_i^s is the amount of data to be sent (e.g., the user input and the current state), and d_i^r is the amount of data to be received (e.g., the execution results). These parameters can be predicted by some forecast algorithms [16].

Given the knowledge of the wireless channel state for the given time span, the function $R(t_i)$ maps the channel state to the effective throughput at each slot t_i, which denotes the true bandwidth considering network delay and packet loss during the data transmission. How to model the wireless channel states and acquire $R(t_i)$ is out of the scope of this paper, which is well studied in the literature. We consider that the energy cost function can be obtained based on network measurements with real hardware. We are then able to calculate the power consumption $E_c(c)$ for the computation of c and the power consumption $E_t(d)$ for the wireless transmission of d. It is worth noting that, in the current formulation, rather than specifying a certain wireless access technology, we employ a generic power model which can be further extended based on various features of different wireless access technologies.

Different network interfaces have different characteristics in power consumption. Moreover, the rapid development of multi-core multi-thread processors has significantly changed the characteristics of the power consumption, not to mention the countless hardware models and various implementations in commercial mobile products. As such, we do not rely on a specific power model here, but consider a generic energy cost function, which allows our formulation and solutions to be easily integrated with different hardware platforms and wireless interfaces [17], [18].

Our goal is to find an optimal schedule for the client to execute (locally) or offload (to the cloud) each task, and minimize the battery power consumption with delay constraints. Let $s_i = (d_i, [t1_i, t2_i], [t3_i, t4_i])$ be the schedule of j_i, where d_i is the offloading decision. $d_i = 0$ indicates that the client decides to execute j_i locally, while $d_i = 1$ indicates that the client offloads j_i to the cloud and receives the execution results. $[t1_i, t2_i]$ and $[t3_i, t4_i]$ are the time intervals for sending data and receiving data, respectively. Note that we only care about the sending and receiving intervals for the tasks that are chosen to be offloaded. If a task is decided to be run locally, we do not consider its schedule. Let F_c be the processing capability of the cloud server that executes the offloaded tasks. Note that the scheduling performs at the client side. The problem can be formulated as finding an optimal schedule $S = \{s_1, s_2, ..., s_n\}$ that minimizes the total battery consumption:

$$E_{total} = \sum_{i=1}^{n} \{(1 - d_i)E_c(c_i) + d_i E_t(d_i^s + d_i^r)\}, \quad (1)$$

and the following constraints should be satisfied:
(1) Causality constraint:

$$\forall i \in [1, n], t1_i \geq t_i^a; \quad (2)$$

(2) Playback continuity constraint:

$$\forall i \in [1, n], t4_i \leq t_i^d; \quad (3)$$

(3) Streaming rate constraint:

$$\begin{aligned} \forall i \in [1, n], \sum_{t_k = t1_i}^{t2_i} R(t_k) \geq d_i^s, \\ \text{and } \sum_{t_k = t3_i}^{t4_i} R(t_k) \geq d_i^r; \end{aligned} \quad (4)$$

(4) Time gap constraint:

$$\forall i \in [1, n], t3_i - t2_i \geq c_i / F_c; \quad (5)$$

(5) Energy saving constraint:

$$\forall i \in [1, n], \text{if } d_i = 1, E_t(d_i^s + d_i^r) < E_c(c_i); \quad (6)$$

(6) Feasibility constraint:

$$\begin{aligned} \forall i \neq j, [t1_i, t2_i] \cap [t1_j, t2_j] = \emptyset, \\ [t1_i, t2_i] \cap [t3_j, t4_j] = \emptyset, \\ [t3_i, t4_i] \cap [t1_j, t2_j] = \emptyset, \\ \text{and } [t3_i, t4_i] \cap [t3_j, t4_j] = \emptyset. \end{aligned} \quad (7)$$

The causality constraint (Equation 2) implies that a job cannot be scheduled before its start time. The performance constraint (Equation 3) follows that computation offloading should not cause performance degradation in terms of user experience even though our goal is to save battery. Meanwhile, the release times and the deadlines can be set to ensure the dependencies among jobs. The data transmission constraint (Equation 4) allows the necessary amount of data to be transferred. The execution constraint (Equation 5) indicates that the cloud server needs enough time to process the offloaded task. The energy saving constraint (Equation 6) guarantees that battery savings can be achieved from the scheduled offloading. Finally, the feasibility constraint (Equation 7) ensures that there is no conflict in the sequential scheduling.

This problem is challenging as each job has two intervals (the sending interval and the receiving interval) to schedule with the restriction of the start time, the deadline, and the gap (processing time) in-between. The criteria of making offloading decision for a specific job is whether there is battery saving while the job can be done before the deadline. This can vary significantly for different schedules under dynamic network environment. Moreover, it is possible for a single job to more than one schedule that can save the same amount of battery power. Considering the scheduling of multiple jobs, the problem becomes even harder, as each scheduled job directly affects the available time slots for the following jobs. For the formulated problem, we have the following theorem:

Theorem 1. *The decision version of the modeled generic offloading scheduling problem is a NP-complete problem.*

Proof: The key idea is to show that the Subset Sum problem is reducible to the decision version of our problem[2]. ■

Although the decision version of this problem is NP-complete, one can think of a dynamic programming algorithm to solve the scheduling problem recursively. For instance, each iteration decides whether to offload the current job or execute it locally to achieve the minimal battery power consumption. Let M be the number of available time slots in the scheduling interval $[t_i^a, t_i^d]$ for job i. In each iteration, there are $O(M^2)$ schedules that need to be checked. For each schedule, constant operations are performed to calculate the battery cost. In the worst case, the naive dynamic programming algorithm cannot find the global optimal until it recurs to the one job case. It is indeed doing the exhaustive search, in which its recursion tree grows exponentially. Thus, it becomes very inefficient with the total run time of $O(M^{2n})$.

To solve the problem efficiently, we propose a greedy heuristic as shown Algorithm 1. It sorts the jobs according to the computation-to-data ratio and schedules the more computation-intensive jobs with higher priorities. As we have discussed earlier, our goal is to save as much battery power as possible by offloading the computation to cloud. Given the network throughput, the amount of data that can be transferred in each time slot and the resulting transmission energy cost are fixed. A higher computation-to-data ratio implies that more battery power could be saved if the corresponding job is

[2]The details of the proof can be found in our technical report at https://www.dropbox.com/s/nw6ei5ia4vj0r3y/version4.pdf?dl=0

Algorithm 1 Greedy Offloading Scheduling

Set the time array T as all available;
Sort the current job queue J by descendant order of $c/(d^s + d^r)$
while true **do**
  Select the first job j_1 in the sorted queue;
  Search the interval $[t_1^a, t_1^d]$ in T for the valid schedules that satisfy all the Constraints (1)-(6)
  if there is no valid schedule **then**
    d_1=0
  else
    d_1=1;
    Select the schedule with the highest battery power saving $E_c(c_1) - E_t(d_1^s + d_1^r)$ (if there are multiple schedules, select the latest one);
    Update T accordingly
  end if
  $J \leftarrow J - \{j_1\}$
  if there is any new job arrived **then**
    Add the new job to J;
    Sort J by descendant order of $c/(d^s + d^r)$
  end if
end while

offloaded. By searching the valid schedule of each job in the order of computation-to-data ratios, the proposed greedy heuristic significantly improves the search efficiency. It can quickly found the near-optimal solution with the run time of $O(nM^2)$. For each of the n jobs, the heuristic checks $O(M^2)$ schedules. When multiple schedules are found with the same amount of highest battery savings, our heuristic algorithm schedules the transmission intervals as late as possible in order to have the minimal impact on the scheduling of later jobs.

V. Performance Evaluation

In this section, we will present our simulation settings and evaluate the performance of our proposed solution through trace-driven simulations.

A. Power Model

We first present the power model that is adopted in the evaluation. It characterizes the power consumption properties of the test device based on our measurements. Since the focus of this work is not on building an accurate power model, we simplify the power model in [18] according to our collected data traces. As we are concerned about the power consumption of function units that are directly related to computation offloading, our model considers mainly the power consumption of CPU and WiFi as shown in Table III.

As we keep the working frequency fixed in our experiments, the CPU power consumption is strongly influenced by its utilization. Regarding the WiFi power consumption, similar to [18], the WiFi interface has low-power state and high-power state in our model. When the WiFi interface is idle or transmitting at a very low rate (maintaining a connection), it stays in the low-power state. If the data rate is high, the WiFi interface works in the high-power state. Other components, such as the screen and other sensors, may also consume a significant amount of power. It should be noted that a more complex power model can achieve higher accuracy, but this model is sufficient for our purpose.

Model	$\beta_u * \text{util} + \beta_{cpu} * \text{CPU}_{on} + \beta_l * \text{WiFi}_l + \beta_h * \text{WiFi}_h + \beta_{others} * \text{Oth}_{on}$		
Category	Variable	Range	Power coefficient (mW)
CPU	util	1-100	19.9
	CPU_{on}	0,1	139
WiFi	WiFi_l	0,1	40
	WiFi_h	0,1	605
Others	Oth_{on}	0,1	740

TABLE III: Power Model

B. Data Traces and Methodology

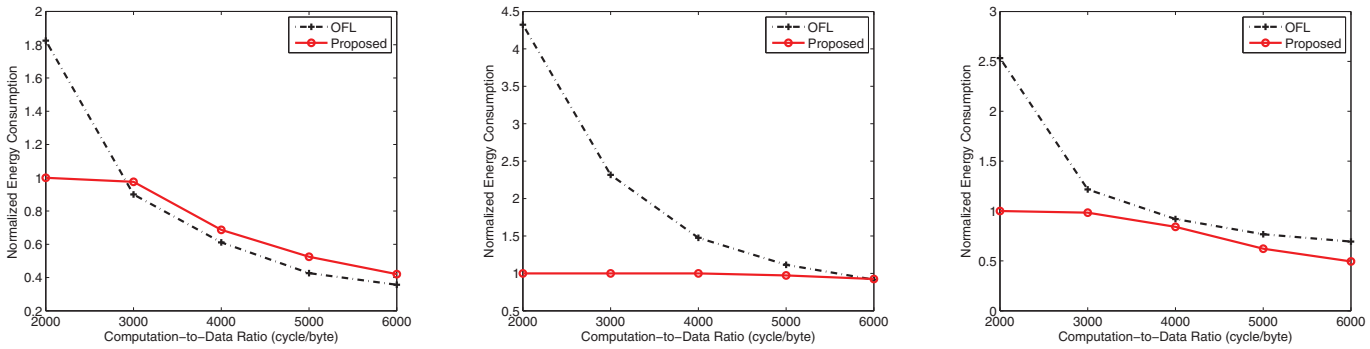
To emulate different network conditions, we collected real-world bandwidth traces from our test devices under three testing scenarios: 1) having a good wireless network connection with bandwidth higher than 10 MB/s, and the device is located near the wireless AP; 2) suffering a bad wireless network connection with bandwidth lower than 3 MB/s, and the device is located far from the wireless AP; 3) experiencing a varying wireless network connection, the device is moved towards and away from the wireless AP periodically. The real-world traces will be used to test the performance of our algorithm under different network conditions.

We simulate the test application with synthetic traces. As shown in previous studies [19], the probability distribution of applications' demand can be described by Gamma distribution in certain cases. In the simulations, we set the job in-coming rate to be approximately 2 jobs per second with the average delay tolerance of 0.5 seconds. We assume that the transferred data during offloading follows normal distribution. By varying each job's computation-to-data ratio, we are able to change the application from data-intensive to computation-intensive. Thus, we can evaluate the performance of the proposed approach with different workloads.

As our algorithm is designed with performance guarantee (constrained by deadlines), we focus on evaluating the energy efficiency of our approach. We also compare the performance of the proposed approach with two baseline schemes: all jobs are executed locally without offloading (LOC); all the jobs are offloaded to the cloud without any local execution (OFL). Both of the two baseline schemes schedule the job execution in a First-In-First-Out (FIFO) manner.

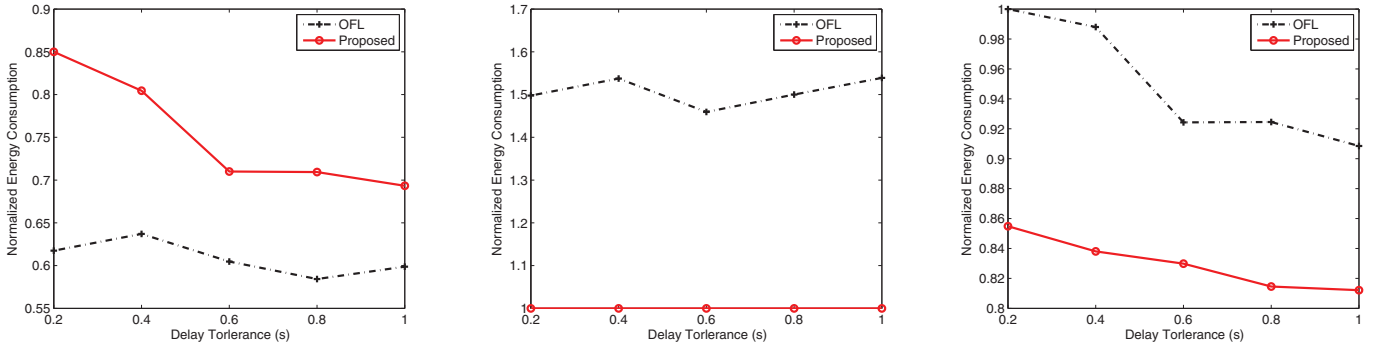
C. Computation-intensive vs Data-intensive

Our first finding is obtained by analyzing our measurement traces. Generally speaking, for computation offloading to be beneficial, the workload needs to perform more than 3000 cycles of computation for each byte of generated data. It is worth noting that this experiment was conducted on our specific test device under stable testing network condition. The result indicates that the characteristics of the offloaded task's workload have a huge influence on the energy efficiency



(a) Energy efficiency in a good channel (b) Energy efficiency in a bad channel (c) Energy efficiency in a varying channel

Fig. 5: Energy consumption normalized by the baseline result of LOC with different computation-to-data ratios



(a) Energy efficiency in a good channel (b) Energy efficiency in a bad channel (c) Energy efficiency in a varying channel

Fig. 6: Energy consumption normalized by the baseline result of LOC with different delay tolerances

of mobile offloading. We then vary the computation-to-data ratio from 2000 to 6000 cycles/byte, and compare the energy efficiency of our solution with the two baselines.

Figure 5 plots the normalized energy consumption of our solution and OFL (both normalized by LOC). Figure 5(a) shows that OFL achieves the best performance when the wireless channel is in a good state. In this case, our solution has similar energy efficiency as OFL. On the other hand, when the wireless channel is becoming worse, our solution outperforms OFL and LOC. Figure 5(b) shows that, if the wireless connection becomes constantly bad, our solution gives up the offloading opportunities and keeps the computation locally. It can reduce the battery usage by making smart offloading decisions in an unstable wireless channel (see Figure 5(c)). Figure 5 also shows that a naive scheme such as OFL can only work well in an ideal condition, which is not able to adapt to the change of network environment. As shown in Figure 5(b) and 5(c), OFL incurs over four times energy consumption compared to LOC in the worst case.

This result also provides us insights about the design principles of mobile offloading systems. It is crucial to differentiate computation-intensive tasks from data-intensive tasks. Figure 5 shows that both OFL and our proposed algorithm can achieve better energy efficiency when the computation-to-data ratio increases, no matter what the wireless channel condition is. With a coarse measurement, a threshold of the computation-to-data ratio can be estimated, where the computation offloading starts saving battery. However, in practice, identifying a task that can benefit from computation offloading is never easy. It depends on many factors, such as hardware model, type of network connection, link quality, type of workloads, and user mobility, etc. Yet a straightforward principle still applies: computation-intensive tasks are more suitable for offloading in all kinds of network conditions. It is not surprising that an extremely computation-intensive task can still save a significant amount of battery power with a poor wireless connection. An extreme example is the chess game. Given the chessboard has 64 positions and each player has exact 16 pieces at the beginning, chess is Markovian, meaning that the game is fully expressed by the current state. Since there are limited number of pieces and possible locations for each piece, a typical wireless packet has enough space to hold the information of the chess game's current state. As computation complexity for searching the best chess moves is extremely high, it makes offloading in a chess game beneficial for most wireless network environment.

D. Delay-sensitive vs Delay-insensitive

Next, we evaluate the performance of our approach and that of other two reference schemes by varying the task delay tolerance (from 0.2 to 1s) with the computation-to-data

ratio set at 4000 cycles/byte. Figure 6 shows the normalized energy consumption of our solution and OFL under different wireless channel conditions. Similar to the previous results, our approach outperforms OFL in a bad or varying wireless channel. It confirms that our proposed approach is superior to the two baseline schemes and can adapt to the change of the wireless link quality. As shown in Figure 6, since the offloading decisions are fixed in OFL, delay tolerance has no significant influence on the energy efficiency of OFL in a good/bad wireless channel. The energy consumption of OFL stays in a relatively stable level (varying within 5%) when the wireless channel is stable, where the jitters simply come from the randomness of our data traces; whereas in a varying wireless channel, when the delay tolerance is relaxed OFL can selectively schedule the data transmission under higher bandwidths, and thus OFL's energy consumption decreases as the delay tolerance grows. It is worth noting that, although OFL can achieve better energy efficiency than our approach in a good wireless channel as shown in Figure 6(a), OFL may violate the deadline constraint for the offloaded tasks given the tight delay tolerance setting, and thus result in significant performance degradation (e.g., frequent interruptions during the video playback).

Unlike OFL, the performance of our proposed solution improves as the tasks have higher delay tolerance. This result indicates that delay-tolerant tasks are more likely to benefit from computation offloading than delay-sensitive tasks. With a higher delay tolerance, the task has higher chances to be offloaded. This is because its data transmission can be deferred to wait for a better network connection. This observation encourages computation offloading for delay-tolerant mobile services. While it is beneficial for chess-like applications, we have to carefully weigh the different factors for delay-sensitive tasks such as realtime video applications. For instance, offloading game execution can provide ubiquitous gaming experience that cannot be achieved on resource-limited mobile devices, but the realtime constraints as shown in Table I must be accommodate. Otherwise, the experience will be poor even with high-resolution scenes.

VI. Conclusions

In this paper, we investigated on power-aware data transmission for computation offloading in mobile cloud. We examined the performance and energy efficiency of computation offloading under various wireless network conditions through measurements in experiments. We discussed the unique features of data communication in mobile cloud offloading that are different from traditional data transmission. Furthermore, we proposed a generic formulation of the data transmission scheduling problem and presented an effective algorithm for scheduling computation offloading in mobile cloud. The simulation results demonstrated that our proposed solution outperforms the existing schemes and can adapt to the dynamic network condition. Finally, we discussed some practical issues, and shed light on the implementation of the mobile computation offloading system.

As for the future work, we are conducting extensive simulations to evaluate and improve this framework with the datasets and execution traces collected from real-world applications. In addition, we plan to develop a prototype system to further verify and evaluate our proposed scheduling scheme, and study the practical challenges in depth. We are also interested in exploring other open issues along this direction, such as designing better service partitioning mechanisms as well as extending our solution to integrate with multiple hardware models and interfaces.

Acknowledgment

This publication was made possible by NPRP grant # [8-519-1-108] from the Qatar National Research Fund (a member of Qatar Foundation). The findings achieved herein are solely the responsibility of the authors.

References

[1] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan, "Bartendr: a practical approach to energy-aware cellular data scheduling," in *ACM MobiCom*, 2010.

[2] L. M. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *IEEE INFOCOM*, 2001.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *ACM MobiSys*, 2010.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *ACM EuroSys*, 2011.

[5] A. Gember, C. Dragga, and A. Akella, "ECOS: practical mobile application offloading for enterprises," in *USENIX Hot-ICE*, 2012.

[6] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *IEEE INFOCOM*, 2012.

[7] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely, "Energy-delay tradeoffs in smartphone applications," in *ACM MobiSys*, 2010.

[8] F. R. Dogar, P. Steenkiste, and K. Papagiannaki, "Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices," in *ACM MobiSys*, 2010.

[9] P. Shu, F. Liu, H. Jin, M. Chen, F. Wen, Y. Qu, and B. Li, "etime: energy-efficient transmission between cloud and mobile devices," in *IEEE INFOCOM*, 2013.

[10] F. Liu, P. Shu, and J. Lui, "AppATP: An energy conserving adaptive mobile-cloud transmission protocol," *IEEE Transactions on Computers*, 2015.

[11] M. Altamimi, A. Abdrabou, S. Naik, and A. Nayak, "Energy cost models of smartphones for task offloading to the cloud," *IEEE Transactions on Emerging Topics in Computing*, 2015.

[12] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in *ACM MobiSys*, 2011.

[13] "Droidfish," http://web.comhem.se/petero2home/droidfish/, available: 2016.

[14] "Osmos," http://www.osmos-game.com/, available: 2016.

[15] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[16] S. Gurun, C. Krintz, and R. Wolski, "NWSLite: a light-weight prediction utility for mobile devices," in *ACM MobiSys*, 2004.

[17] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, "NAPman: network-assisted power management for wifi devices," in *ACM MobiSys*, 2010.

[18] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *IEEE/ACM/IFIP CODES*, 2010.

[19] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *ACM SIGMETRICS*, 2001.