國立清華大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science

College of Electrical Engineering and Computer Science

National Tsing Hua University

Master Thesis

降低行動裝置上大型時間尺度傳輸排程的資源消耗

Reducing Training Overhead of Large Time-Scale Transfer Scheduling for Mobile Devices

林庭安

Ting-An Lin

指導教授：徐正炘 博士

Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 102 年 6 月

June, 2013

國立清華大學
資訊工程研究所

碩士論文

降低行動裝置上大型時間尺度傳輸排程的資源消耗

林庭安 撰

102
6

Acknowledgments

I would like to express my gratitude to all those who helped me during the two years in National Tsing Hua University.

First, I would like to appreciate Yichuan Wang and Prof. Xin Liu from UC Davis. Thanks for your contributions in our joint work, it's a good experience to cooperate with you. In addition, the results of our joint work play an import role in my thesis. Second, I would like to appreciate my adviser: Prof. Cheng-Hsin Hsu. Without your instruction in this two years, I could not complete my thesis. Last, I would like to appreciate my parents. Thanks for your support so that I can get B.S. and M.S. degrees.

致謝

在此，我想要感謝所有在我在清華大學的這兩年裡幫助過我的人。

我首先要感謝來自UC Davic的王一川與劉昕老師， 謝謝你們在我們的合作計畫中的所有貢獻， 和你你們合作是一個非常好的經驗。 我們的合作計畫的成果在我的畢業論文裡扮演了重要的角色。 接著，我要感謝我的指導教授徐正炘老師， 謝謝你在這兩年之中的指導，讓我可以完成我的畢業論文。 最後我要感謝我的父母，謝謝你們的支持，讓我可以完成大學與碩士班的學業。

中文摘要

在這篇論文中，我們專注於大型時間尺度(數分鐘到數小時)下的傳輸排程，它提供了更大的傳輸效能改進空間，這是我們與大多數在頻道感知排程方面的研究不同之處。 我們在Android平台上設計、實作和驗證了一個基於馬可夫決策理論的框架用來分析使用者行為並排程傳輸。 我們的模擬結果顯示，現實中的行動裝置裝使用者可以受益於我們的框架。 舉例來說，50％的使用者享受到20％-90％的傳輸速度提升與15分鐘的平均延遲，當截止時間為40分鐘時。

此外，我們的量化與降低產生排程演算法參數的資源消耗。 我們指出了產生排程演算法參數的最佳使用者資料長度為30天。 我們用不同的聚類演算法對使用者進行分組，以減少產生排程演算法參數的資源消耗。 藉由使用從真實的使用者收集到的資料，我們指出了聚類演算法最佳參數。 同時，我們的聚類演算法，可以減少產生排程演算法參數的資源消耗，但不會損失太多的效能。 當使用了我們的演算法，可以節省產生排程演算法參數的時間高達59.9％，但只導致了小於18％的效能下降。

Abstract

In this thesis, we focus on large time-scale scheduling of mobile data transfer, e.g., in minutes or hours. Such large time-scale provides significantly more room for performance improvement in real-life scenarios, which differs our work from most existing channel-aware scheduling studies.

In particular, we design, implement, and evaluate a framework for profiling and scheduling based on Markov decision theory, using the Android platform. Our trace-driven simulations show that mobile users in real-life scenarios can benefit significantly from our framework. For example, 50% of mobile users will enjoy 20%-90% throughput improvement with a deadline guarantee of 40 minutes and an average delay of 15 minutes.

In addition, we quantify and reduce the overhead of generating model parameters of the proposed scheduling algorithms. We determine the best training window size: 30 days. We adopt various clustering algorithms to group users in order to reduce training overhead. We empirically determine the best system parameters of the clustering algorithms using real traces. Our clustering algorithms reduce the training overhead without sacrificing too much performance: it saves up to 59.9% of training time while incurring $<$18% performance degradation.

Contents

List of Figures

List of Tables

Chapter 1

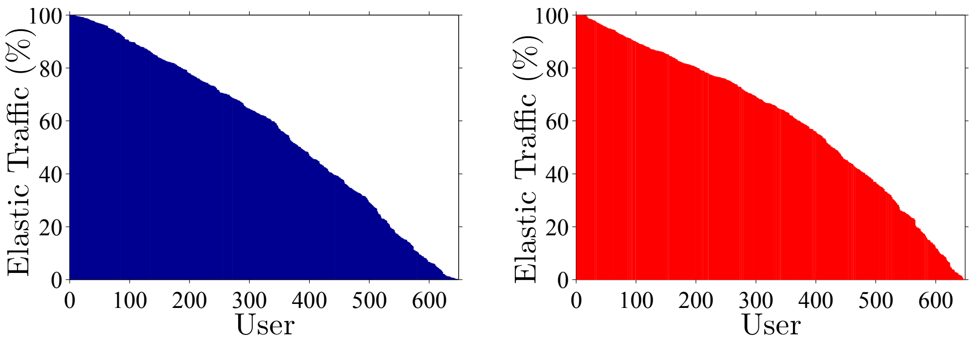
Introduction

Consider the following scenario, we saw an interesting show on the way to the subway station and recorded a video of it using smartphone. The phone automatically uploaded the video to social network such as Facebook, then we can share it with our friends. Unfortunately, the automatic upload may take more than 30 minutes and consume half of the battery, because the video size was large and the network in subway station was crowded. In addition, it cost a few dollars because the telecom provider no longer provides unlimited cellular dataplan.

Ideally, the upload application could be more intelligent. For example, once detecting a large video upload request, it could provide options like "Upload now", "Within 10 minutes", "Within 1 hour", and "WiFi only". Suppose that we chose the 1-hour option, then the upload possibly happened 30 mins later when we stopped by a coffee shop where the network condition was much better. Even better, imagine the telecom providers provide an option like "Delay for 30 mins and enjoy a 20% discount", similar to the trial conducted with AT&T customers in Princeton using dynamic time-dependent pricing [7].

There are a few key components in this example. First, mobile network access is opportunistic, due to user mobility and wireless network dynamics. Therefore, the network conditions, including network availability, network load, signal strength, and transfer speed are various in different times and locations.

Second, a large portion of mobile traffic, such as email and user-generated multimedia contents, can tolerate delay in different time scales, ranging from seconds to hours. We refer to such traffic as *elastic*. To show the extent of such traffic, we collected data traffic profiles from ~ 700 real Android users for up to 5 months. A coarse classification on the applications shows that more than 50% users generate at least 65% (uplink) and 70% (downlink) elastic traffic, plotted in Fig. 1.1. The analysis is similar to the observation in [16], where analysis of logs from a cellular network shows that there is a nontrivial time difference between multimedia content generation and upload: more than 55% multimedia contents are uploaded 1+ day after the creation time.



(a) (b)

Figure 1.1: The elastic traffic ratio on: (a) uplink and (b) downlink.

Third, for good user experience there should be an application-specified **deadline** for content transfer. Such a deadline guarantee provides predictability user experience. For example, once one chooses the ”1-hour” option, the user needs not to worry about it anymore knowing that the job will finish in at most an hour.

Existing work has studied delay tolerance and channel variation for improved battery performance and resource utilization, e.g., in [4, 14, 15]. Most of the existing works consider small time-scale, from seconds to minutes. However, they do not leverage the much larger optimization room that is only achievable via large time-scale scheduling. In this paper, we consider a much larger scale, from minutes to hours.

When the users allow a large delay, e.g., 30 mins, the users may visit different locations and experience diverse network conditions. Thus, the network condition will exhibit highly *non-stationary* patterns. Fig. 1.2 shows such an example where the signal strength varies with time. Using immediate past measurements at the “current time” to predict future network condition until the “deadline” will clearly be highly inaccurate. This figure also emphasizes the large optimization room when we consider large time-scale scheduling.

To overcome such limitation, we study a new approach of network optimization: we schedule network access base on large time-scale user behavior and network condition statistics. We propose and implement the User-Profile-Driven Adaptive TransfEr (UPDATE) framework, and make the following contributions:

- We develop an optimal decision framework to schedule elastic traffic with delay guarantees based on user profile.
- The framework supports different optimization objectives including minimizing (i) network resource consumption, (ii) energy consumption, and (iii) access cost.

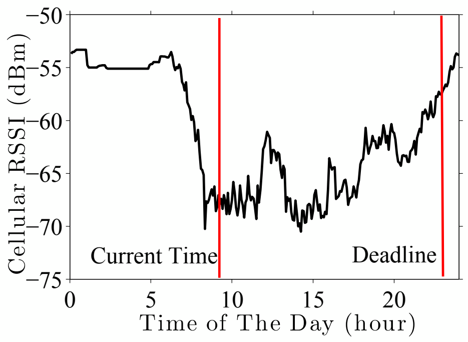


Figure 1.2: The cellular signal strength of a sample Android user, who has much better reception at home than in the office.

- We conduct realistic performance evaluations based on real-life traces generated by the general public. Results show that a majority of the regular mobile users can benefit from the UPDATE framework and significant diversity among regular users in all aspects.
- We study the overhead of training the model parameters of our framework and derive optimal clustering schemes to reduce those overhead via extensive trace-driven simulations.

Chapter 2

Related Work

2.1 Large Time-scale Transfer Scheduling

The idea of leveraging delay tolerant content transfer has been studied in the literature. In [4], the authors used WiFi network whenever possible to offload data from 3G connections, which is achieved by analyzing recent WiFi availability to predict the future availability. In [14], the authors use a Lyapunov function based approach to schedule delay-tolerant traffics based on network condition and queue length. We note that Lyaponov optimization can balance the tradeoff between delay tolerance and performance improvement by adjusting its control parameter. It achieves asymptotical optimality without statistical information. In comparison, we consider an optimal stopping problem that is more suitable under deadline constraint and statistical information. In [4, 14], they use short history to predict future network condition, which is valid for a short time scale, say minutes.

Schulman et al. [15] used location service to derive the user paths, which are then leveraged to predict future network condition. User paths can be considered as a special kind of user profile although still in small time scales (minutes level).

Our work complements existing work by focusing on large time-scale (minutes to hours) for further performance improvement. Because of the large time scale, we propose to use historic user profiles that enables us to provide statistical prediction, which is different from existing work. Furthermore, deadline is not considered in the previous work, which is the focus of this work and thus its technical contribution. Last, previous efforts have used traces from the experiments conducted by researchers, and our evaluation is based on real-life traces from the general public. In summary, our work compliments existing approaches in its large time scale, deadline-driven objective, and its real-world evaluation.

2.2 Mobile User Clustering

There are number of existed works on mobile user behavior and mobile user clustering. Some studies are about location and mobility of mobile users [6, 10, 11]. Bayir et al. [6] proposed a framework to capture cellphone users mobility. In [10, 11], the authors proposed user mobility model to predict the mobility of mobile users. Compared with our work, those works studied location and mobility of mobile users.

There are works studied the relation of different user mobility [9,18,19]. Hsu et al. [9] proposed an user similarity metric based on a matrix representation of user mobility profiles and its decomposition to reduce complexity of user similarity computation. Then, they cluster users based on the proposed similarity metric to reduce the traffic load of a campus WLAN. Ying et al. [18] proposed a potential friends recommendation algorithm based on the GPS trajectories of mobile users. Zheng et al. [19] proposed to mine interesting locations and classical travel sequences from the GPS trajectories traces of a specific region. Compared with our work, those works studied the mobile users from a specific region.

There are works studied mobile user behaviors except mobility, e.g., application usage [13, 17]. In [13], the authors proposed to prefetch mobile advertisement based on the application usage prediction. In [17], the authors studied the prediction of mobile application usage and proposed to prelaunch application on mobile devices to reduce the application launching time. Different to our work, the authors did not consider the network access behavior of mobile users.

In addition, there are existed works consider multiple mobile user contexts, e.g., user interactions and locations, when mining mobile user behaviors. For example, Ma et al. [12] proposed an approach to mine user behavior patterns and leverage the behavior patterns for discover similar mobile users.

Different from those existing studies, our goal is to reduce the overhead of training prediction models by cluster users with similar behavior. To our best knowledge, using clustering to reduce the overhead has not been studied before.

Chapter 3

Large Time-scale Transfer Scheduling

In this chapter, we introduce the design and implementation of our proposed large time-scale transfer scheduling framework. Then, we evaluated our proposed framework via extensive traces-driven simulations with a large amount of traces collected from real mobile device users.

3.1 UPDATE Framework

The UPDATE framework runs on mobile devices and consists of two components: (i) *profiler* and (ii) *resource manager*, as illustrated in Fig. 3.1. The profiler is an application-layer program that collects user profiles, which are timestamped log files, including network condition and application/data usage. The profiler also monitors storage usage and feeds the resource manager the most updated profile information. Table 3.1 summarizes the contexts collected by the profiler.

The resource manager monitors the content transfer requests and transmission. It consists of (i) *policy generator*, (ii) *scheduler*, and (iii) application API. According to the desired objective, e.g., to minimize energy consumption, the policy generator generate scheduling policies based on user profiles. The policy generator can run on mobile devices and it can also be offloaded to the cloud. Mobile applications communicate with the UPDATE framework via the API and submit content transfer requests. Upon receiving such a request, the scheduler schedules the transmission according to the scheduling policies.

3.2 Markov Decision Process

Markov Decision Process (MDP) is a well-known mathematical framework for modeling decision making. The MDP can be defined as a 4-tuple: $(S, A, P.(\cdot,\cdot), R.(\cdot,\cdot))$. In the

Table 3.1: The Contexts Being Profiled on Smartphones

Context	Profiling Type	Period (min)	Profiling Level ($\geq$)
WiFi Connectivity	Event-driven	-	Default
3G Signal Strength	Event-driven	-	Default
Activity Information	Periodical	5	Verbose
Task Information	Periodical	5	Verbose
Battery Level	Periodical	5	Baseline
Network Throughput	Periodical	5	Default
Application Traffic Amount	Periodical	5	Default
GPS Location	Periodical	30	Verbose
Neighboring WiFi AP Information	Periodical	30	Verbose
Neighboring Cell Tower Information	Periodical	5	Verbose

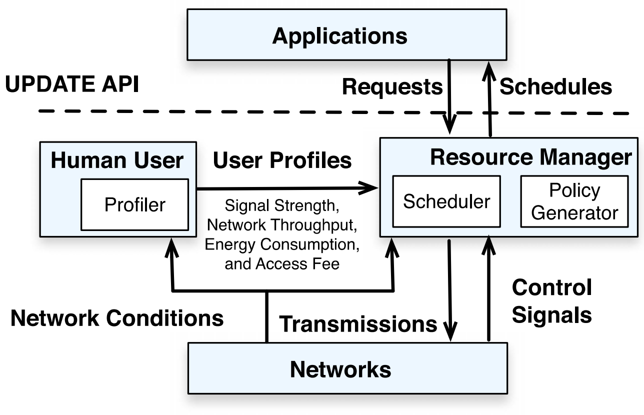


Figure 3.1: The proposed UPDATE framework.

notation: (i) S is the set of possible system states, (ii) A is the set of allowable actions, (iii) $P_a(s, s') = Pr(s_{t+1} = s'|s_t = s)$ is the probability that when the decision maker chooses action a in state s at time t, then the system will transfer to state s' at time $t + 1$ and (iv) $R_a(s, s')$ is the reward of choosing action a in state s, then the system transfers to state s'. The objective of MDP decision problems are to find out an optimal policy π: a function that specifies the action $\pi(s)$ should be chosen when the system in state s. The policies are usually solved by value iteration or policy iteration method to provide what action should be chosen in each state, so that it can get the largest total reward.

3.2.1 Optimal Stopping Problem

In particular, our problem can be formulated by an optimal stopping problem. In optimal stopping problem, the system is an uncontrolled Markov chain with states S'. In each decision epoch, there has two actions available for the decision maker: to stop or to continue. Once he decides to stop in state s, he can get the reward $g_t(s)$. If he decides to continue, he incurs a cost $f_t(s)$ and the system evolve until the next decision epoch. The problem must be finite horizon, in which case $h(s)$ represents the reward if the system reach the last decision epoch N without stop. When the system stop, it will stay at stop state and with zero reward. The objective of optimal stopping problem is to maximize the total reward.

3.3 Scheduling Algorithms

In this section, we introduce our proposed scheduling algorithms based on Markov decision theory. All of the proposed algorithms are deployed into our framework.

3.3.1 Scheduling Model

Our framework assumes a time slotted system, where the timeslot size can vary from seconds to minutes, depending on the time horizon of the deadline and complexity tolerable. When a transfer request arrives, the scheduler is triggered. The request specifies a deadline, named *horizon* N, by which the data transfer must be completed.

At each timeslot, the scheduler makes a *decision* $D_t \in \{\text{Wait}, \text{Transfer}\}$ depending on the current transmission cost and future estimates. Let X_t ($t \in [1, N]$) be the *transfer cost* at slot t and waiting costs nothing. We assume the transmission cost is proportional to file size, and thus files of different sizes share the same decision policies. Let V_t be the *optimal cost* to transfer data between time slot t and N when an optimal schedule is applied. V_t can be calculated using the statistics of X_t, derived from the user profiles.

3.3.2 Optimal Stopping Scheduling (OSS)

The OSS algorithm is based on Markov decision theory, in particular, optimal stopping. The objective is to minimize the expected transfer cost under the deadline constraint. The principle of optimality applies as follows. If transfer cost X_t is less or equal to the expected optimal future transmission cost, transfer at the time slot t, otherwise wait until time slot $t+1$. Formally, the optimal scheduler is written as:

$$D_t = \begin{cases} \text{Transfer}, & X_t \leq E(V_{t+1}|X_t); \\ \text{Wait}, & X_t > E(V_{t+1}|X_t). \end{cases} \tag{3.1}$$

Here, the value of $E(V_t|X_{t-1})$ can be obtained by backward induction:

$$\begin{aligned} & E(V_N|X_{N-1}) = E(X_N|X_{N-1}); \\ & E(V_t|X_{t-1}) = \\ & \sum_c P(X_t = c|X_{t-1}) \min(c, E(V_{t+1}|X_t = c)). \end{aligned} \tag{3.2}$$

The algorithm presented in Eq. (3.1) and (3.2) is general, but may come with some limitations. First, we need to accumulate enough samples to accurately derive $P(X_t|X_{t-1})$ for all t and X_{t-1}. Hence, the OSS algorithm requires a long user profiles to derive model parameters. Second, the computational complexity can be high as the horizon increases as shown in Sec. 3.4. This is a potential concern for mobile users.

3.3.3 Lightweight Optimal Stopping Scheduling (OSS$_\mathrm{L}$)

To alleviate the limitation of the OSS algorithm, we propose a simplified scheme using the light model where X_t depends only on time. In this case, the optimal decision is simplified as:

$$D_t = \begin{cases} \text{Transfer}, & X_t \leq E(V_{t+1}) \\ \text{Wait}, & X_t > E(V_{t+1}) \end{cases}, \tag{3.3}$$

where

$$\begin{aligned} E(V_N) =& E(X_N); \\ E(V_t) =& P(X_t > E(V_{t+1}))E(V_{t+1}) \\ & + P(X_t \leq E(V_{t+1}))E(X_t|X_t \leq E(V_{t+1})). \end{aligned} \tag{3.4}$$

3.3.4 Batched Optimal Stopping Scheduling (BOSS)

Next, we consider a more complex scenario where multiple jobs arrive sequentially from the application layer. In this case, we need to consider *overhead*. Overhead occurs because of network setup, signaling, and tail effect. For example, in UMTS networks, the release of radio resources is controlled by inactivity timers. It's also known as the tail time can be up to 17 seconds. That is a significant waste of network resource when severe for small network traffic, prevalent in today's mobile applications.

Let Q denote the scheduler queue at the beginning of the timeslot; Q^+ the queue after job arrivals in this timeslot; and Q^- the queue after transmitting the job with the closest deadline.

We call a timeslot *active* if one or more content transfers are scheduled, otherwise the timeslot is *inactive*. In an active slot, batching additional jobs incurs no additional overhead. Denote A_t^Q the expected cost when using the optimal policy, starting from an active timeslot t, and C_t^Q that of an inactive timeslot. Let T be the total amount of time we run the scheduling algorithm. The expected costs with different actions in timeslot $t-1$ are written as follows. If no job is transmitted in the current timeslot $t-1$ and the scheduler goes to next timeslot t, then

$$C_{t-1}^Q = C_t^{Q^+}, A_{t-1}^Q = C_t^{Q^+}. \tag{3.5}$$

If the scheduler schedules the job with the earliest deadline in the queue and stays in the current timeslot $t-1$, then

$$C_{t-1}^Q = O + X_{t-1} + A_{t-1}^{Q^-}, A_{t-1}^Q = X_{t-1} + A_{t-1}^{Q^-}. \tag{3.6}$$

1) In an inactive timeslot $t-1$, if $X_{t-1} > C_t^{Q^+} - A_{t-1}^{Q^-} - O$, then the decision is to transmit no requests, and go to next timeslot; otherwise, the decision is to transmit the first request, and stay in the current timeslot.
2) In an active timeslot $t-1$, if $X_{t-1} > C_t^{Q^+} - A_{t-1}^{Q^-}$, then the decision is to transmit no more request, and go to the next timeslot; otherwise, the decision is to transmit the request with the closest deadline, and stay in the current timeslot.

We note that the complexity of BOSS is very high because the number of states is prohibitively large due to the combinatorial nature of the possible deadlines and state transmissions. In fact, BOSS cannot be used for deadlines more than three time slots long as shown in our evaluation.

3.3.5 Lightweight Batched Optimal Stopping Scheduling (BOSS$_\text{L}$)

The BOSS$_\text{L}$ is a light version of BOSS where X_t is only dependent on the time. Given the BOSS algorithm, we can use backward induction to calculate C_t^Q and A_t^Q as:

$$\begin{aligned} C_{t-1}^Q = {} & P(X_{t-1} > C_t^Q - A_{t-1}^{Q^-} - O)C_t^{Q^+} \\ & + P(X_{t-1} \le C_t^Q - A_{t-1}^{Q^-} - O)(X_{t-1} + A_{t-1}^{Q^-} + O); \end{aligned} \tag{3.7}$$

and

$$\begin{aligned} A_{t-1}^Q = {} & P(X_{t-1} > C_t^Q - A_{t-1}^{Q^-})C_t^{Q^+} \\ & + P(X_{t-1} \le C_t^Q - A_{t-1}^{Q^-})(X_{t-1} + A_{t-1}^{Q^-}). \end{aligned} \tag{3.8}$$

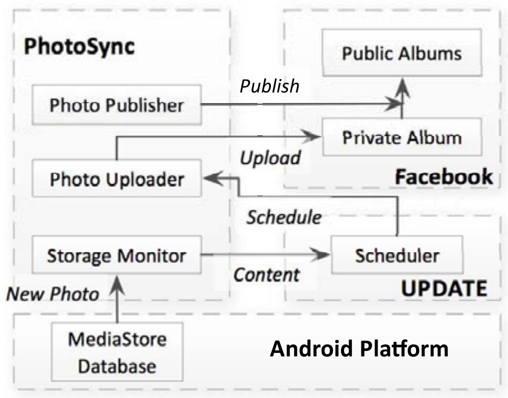


Figure 3.2: The architecture of PhotoSync.

Finally, at the end of the horizon T, we have:

$$C_T^Q = O + |Q|E(X_T), \tag{3.9}$$
$$A_T^Q = |Q|E(X_T). \tag{3.10}$$

3.4 Trace-Driven Simulations

To evaluate our proposed framework, we implement trace-driven simulators and run simulations with a large amount of traces collected from real mobile device users. In this section, we introduce the simulation setup and report the simulation results.

3.4.1 An UPDATE-enabled Application: PhotoSync

PhotoSync is an Android application which schedules photo uploads to Facebook automatically. PhotoSync has three components: *storage monitor*, *photo uploader*, and *photo publisher*, as illustrated in Fig. 3.2. The storage monitor is a background service watching for new photos taken by users. Once a new photo is detected, the storage monitor obtains the new photo. Then, the storage monitor calls the UPDATE API to submit the new photo to the UPDATE scheduler queue. The UPDATE scheduler schedules the transmission and notifies the photo uploader. Upon instructed by the scheduler, the photo uploader uploads photo to a private photo album. Users can review the photos and publish selected photos to their friends or to the public by the photo publisher.

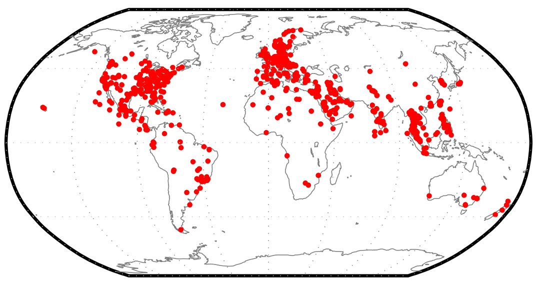
3.4.2 Profiling Analysis

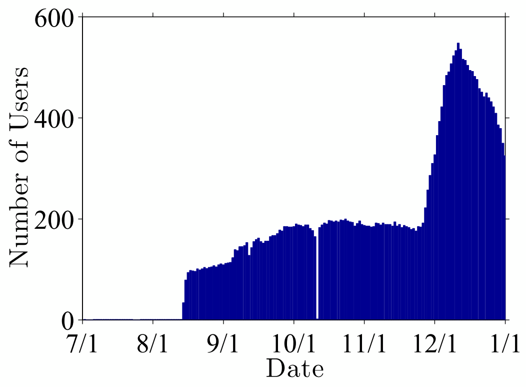
We published Photosync with our profiler on Google Play [1] in July 2012. There have been more than 10,000 downloads in the first eight months. Among them, about 1700+

users participate in our data collection. After filtering out the profiles with zero-length and corrupted data, we ended up with the profiles from $\sim$ 700 users. Among them, we have more than 100 users with 30+ days of profiles, which are used in our performance evaluations.

We first compute the per-application traffic amount across all mobile users. It shows that the top 50 applications contribute $\sim 80\%$ of the traffic, and the top 10 applications contribute $\sim 60\%$ of the traffic. We also roughly classify the applications into two groups, applications that generate: (i) elastic and (ii) real-time traffic. For the top ten downlink apps, we consider dropbox, social network content pre-fetching, application update as elastic; and android browser and youtube as real-time. For uplink traffic, we considered dropbox, social network photo backup as elastic and others real-time including browser, messaging, and video conferencing. Based on this classification, we compute the per-user elastic traffic fraction of the top 50 applications (in each direction). There are on average 65% (uplink) and 70% (downlink) elastic traffic, which shows the potential of the UPDATE framework. We had mentioned that this is coarse classification, and delay tolerance may differ from user to user. However, we note that the UPDATE framework can be applied to the general scenario of different delay tolerance.

We also analyze the profiles in spatial and temporal domains and show the results in Fig. 3.3. Fig. 3.3(a) gives the GPS locations of users who use our application. We can see that the users are from worldwide. Fig. 3.3(b) presents the number of users who upload profiles in each day. We find that the longest profile length is 136 days and there are up to 500+ users in some days.From this analysis, we firmly believe that our trace-driven simulations are representative.


(a)



(b)

Figure 3.3: User distributions in (a) spatial and (b) temporal domain.

3.4.3 Profiling Overhead

We next quantify the energy overhead of the profiler. We install the profiler on an HTC Sensation XE smart phone, and connect it to an Agilent 66331D power meter [2] to measure the average power consumption with and without the profiler for an hour. To be conservative, we killed all other tasks on the phone when measuring the power consumption. We repeat this experiment five times, and report the average power consumption in Table 3.2. The average power overhead of the profiler is merely 2.94 mW, or 6% of the total power consumption without any other applications.

We also install the profiler on four HTC One X smart phones. We fully charge the smart phones, and measure the time of depleting the battery with and without the profiler, and *without any other applications*. Fig. 3.4 shows the battery lifetime, where, in all cases, the battery lifetime is longer than 4.5 days, even with the profiler running. Running the profiler only consumed 11% of the idle energy, which is a small percentage of the actual mobile energy consumption, on average.

Table 3.2: Profiler Power Consumption

Setup	Average (mW)	Min (mW)	Max (mW)
Baseline	48.9	47.1	50.5
UPDATE	51.84	46.9	56.3

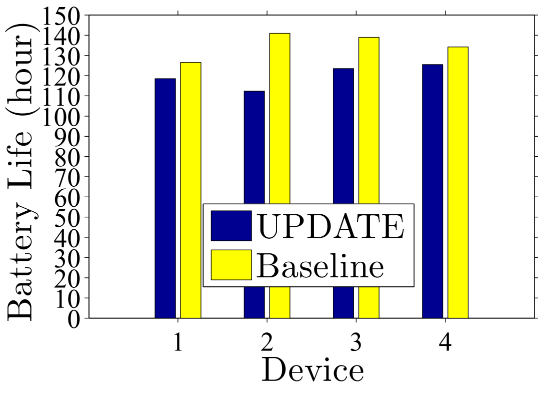


Figure 3.4: Battery life with/without our profiler.

3.4.4 Energy Model

The UPDATE framework is general, and may work with any state-of-the-art energy models to minimize energy consumption. In this paper, we focus on the communication energy, as other parts of energy consumption are not affected by scheduling. We adopt and

Table 3.3: Current Drawn by Network Interfaces under Different RSSI Values

WiFi Network Interface					
RSSI (dBm)	-81.24	-71.24	-60.94	-46.60	-36.6
Current (A)	0.28	0.26	0.25	0.24	0.23
Cellular Network Interface					
RSSI (dBm)	-91.65	-86.14	-73.16	-67.05	
Current (A)	0.33	0.26	0.22	0.21	

slightly modify the model proposed in [5], in which the energy consumption is modeled by a linear function on the transfer time. The energy overhead due to network interface state changes is captured by the ramp and tail energy. The ramp energy refers to the energy consumed by waking up a network interface, and the tail energy is consumed by the timeout period before a network interface is put into asleep. In cellular networks, the tail energy dominates the energy overhead, while in WiFi networks, the ramp energy dominates.

To derive energy model parameters, we set up an experiment to measure the current drawn by the network interfaces of an HTC Sensation XE phone, using an Agilent 66321D power meter [2]. In particular, we instruct the phone to transmit large files and take current readings. Our measurement results indicate that the current is *not* constant under different signal strength in RSSI, as assumed in existing models [5]. Instead, the current depends on the RSSI. We place the phone in locations with different RSSI values, and compute the mean current of each location based on 100,000 samples. Table 3.3 gives the resulting current values, which are used in our trace-driven simulations. We use the same setup to measure the WiFi ramp and cellular tail power consumption, which are 1.02 and 0.74 watts, respectively.

3.4.5 Simulators Implementation

We have implemented a trace-driven simulator in Matlab, and run it on a Linux server with a 2.6 GHz AMD CPU. Within the simulator, we have implemented the four proposed algorithms: OSS, OSS_L, BOSS, and BOSS_L. We have also implemented a baseline algorithm called instant (INS), which transfers a content upon its arrival, and an offline algorithm called optimal (OPT), which assumes the availability of all future network conditions and always makes the best decisions. The INS algorithm mimics the content transfer policy used by most current applications, and the OPT algorithm gives us a performance upper bound, although not achievable. We have also implemented two state-of-the-art algorithms [14, 15] for comparisons.

We consider the traffic from the seven elastic applications we had discussed as the contents to be scheduled by the algorithms, and the real-time traffic to be immediately transferred. In our simulations, we optimize for throughput with wireless ramp/tail overhead of 8 secs, profile length of 22.5 days, time slot length of 5 mins, and deadline of 40 mins (8 time slots).

3.4.6 Simulation Results

We first conduct the simulations of single-job problems, and compare the performance of INS, OSS, OSS_L, and OPT. We report the sample results of the uplinks.

Complexity of f Single-job Scheduling. We study the complexity of the OSS and OSS_L algorithms under different deadlines of 2, 3, 4, 8, and 16 time slots. We randomly choose 10 mobile users, and run the two algorithms with their profiles. We report the average time and memory used for generating the policy tables in Table 3.4. This table shows that the time and space complexity of the OSS algorithm is much higher: it takes almost an hour and consumes almost 16 GB memory to generate the policy tables. In contrast, the OSS_L algorithm terminates in less than 200 msec and with 18 KB memory. This shows that the OSS_L algorithm can be run on even low-end smartphones, while the OSS algorithm has to be offloaded to a cloud server. We emphasize that the policy table generated by the OSS algorithm is fairly compact: it is up to 2.53 MB (with a deadline of 16), which is insignificant to smartphones. Hence, we recommend the OSS algorithm for mobile users with access to cloud servers, and the OSS_L algorithm for other mobile users.

Table 3.4: Complexity of OSS and OSS_L

Time Complexity (sec)					
Deadline	**2**	**3**	**4**	**8**	**16**
OSS	24.65	63.33	102.56	457.20	3403.38
OSS_L	0.02	0.03	0.05	0.08	0.15
Memory Requirement during Computation					
OSS (GB)	0.25	0.56	0.99	3.96	15.82
OSS_L (KB)	2.25	3.38	4.5	9	18

Throughput Improvement. We present the results when optimized for throughput in Fig. 3.5. Fig. 3.5(a) presents the average throughput normalized to that of the INS algorithm from 16 users with the most complete profiles. This figure shows that OSS and OSS_L algorithms lead to throughput improvement for almost all sample mobile users, up to about 2.4 and 1.9 times, respectively. Fig. 3.5(b) shows the CDF of the normalized throughput of all users. We observe that OSS and $\mathrm{OSS_L}$ achieve comparable throughput improvement: more than half of mobile users achieve 20+% throughput improvement. We note that they both outperform BAR (Bartendr [15]) and SAL (SALSA [14]). Last, Fig. 3.5(c) shows the CDF of per-user average delay. We note that (i) all the transfers are done by the deadline as required; (ii) more than 40% mobile users complete their transfers in 10 minutes, much earlier than the user-specified deadline; and (iii) the average delays are 15.11 minutes for OSS and 13.85 minutes for $\mathrm{OSS_L}$, respectively (7.75 and 20.52 minutes for BAR and SAL). In summary, Fig. 3.5 clearly demonstrates the benefit of OSS and $\mathrm{OSS_L}$ algorithms.

Network Load Reduction. We have shown that the OSS and $\mathrm{OSS_L}$ algorithms help the mobile devices to achieve higher end-to-end throughput. Higher throughput results in shorter transfer time, and thus lower network load for the mobile Internet service providers. We next configure the UPDATE framework to explicitly reduce the cellular network load. We define a network load metric for the cellular network symbol/sec, which is based on the cellular network spectrum efficiency (in bits/symbol) and the measured throughput (in bps). We follow the LTE-Advanced standards [3], and map the measured RSSI to the proper Modulation and Coding Scheme (MCS) modes.

We plot the resulting network load in Fig. 3.6. Fig. 3.6(a) shows the network load of 16 mobile users. This figure reveals that the OSS and $\mathrm{OSS_L}$ algorithms reduce the network load, compared to INS. Fig. 3.6(b) shows that the reduction could be as high as 75%. We also plot the CDF curves of overall network load reduction in Fig. 3.6(c), which depicts that the $\mathrm{OSS_L}$ algorithm reduce the cellular network load: more than 40% mobile users cut their network resource consumption by half or more. We note that this improvement includes the benefit of both transmitting in better cellular conditions and offloading to WiFi networks within the deadline. Last, on average, BAR and SALSA consume comparable network resources than $\mathrm{OSS_L}$ and OSS, respectively.

We then conduct the simulations of multiple-job problems, and compare the performance of INS, BOSS, and $\mathrm{BOSS_L}$. We do not consider OPT due to the its high complexity.

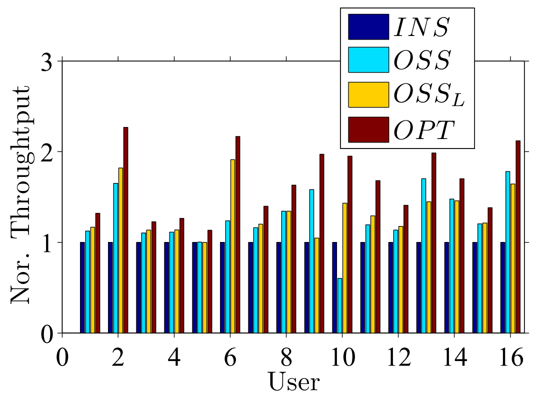
Complexity of Multiple-job Scheduling. We study the complexity of the BOSS and $\mathrm{BOSS_L}$ algorithms, using a similar setup in single-job scheduling. We cannot run the BOSS algorithm with deadlines ≥ 4 because of its complexity. Table 3.5 presents the average time and space used for generating the policy tables with different deadlines. This table shows that the BOSS algorithm is too complex to be practically useful and thus ignored in the rest of the section. With a deadline of 16, we observe that $\mathrm{BOSS_L}$: (i)takes 2.84 hours to complete and (ii) has a policy table of 129.6 MB, which is borderline manageable to high-end smartphones.

Table 3.5: Complexity of BOSS and BOSS$_L$

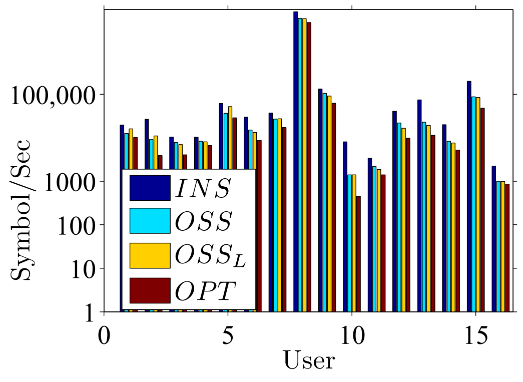
Time Complexity (sec)					
Deadline	**2**	**3**	**4**	**8**	**16**
BOSS	2200.23	8403.63	-	-	-
BOSS$_L$	0.65	1.15	2.5	40	10229
Memory Requirement during Computation					
BOSS (GB)	3.95	15.82	-	-	-
BOSS$_L$ (KB)	9	18	36	576	147456

Energy Consumption Reduction. We report the communication energy consumption normalized to that of the INS algorithm in Fig. 3.7. Fig. 3.7(a) presents sample results from 16 mobile users, which shows that the BOSS$_L$ algorithm outperforms the OSS$_L$ algorithms for majority of the mobile users. This is because the BOSS$_L$ algorithm batches content transfers to reduce ramp/tail energy overhead. We note that BOSS$_L$ ignores file size in calculating its policy, and thus not always optimal, which explains the reason that occasionally OSS$_L$ outperforms it.

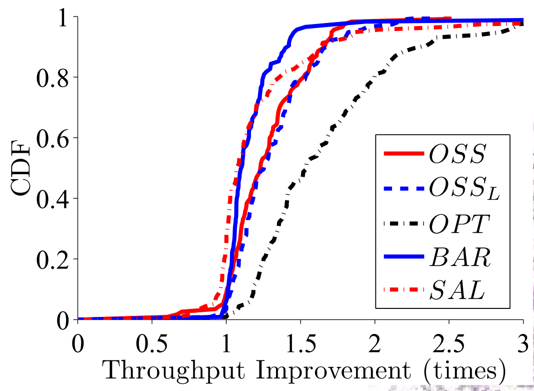
Fig 3.7(b) reports the overall normalized energy consumption as CDF curves. This figure clearly shows the benefits of batching: more than half of the mobile users achieve more than 25% of energy saving with BOSS$_L$, while less than half of the mobile users achieves more than 5% of energy saving with OSS$_L$, and even less for BAR and SAL. We note that in our traces, a large portion of the data jobs are small. Therefore, without batching (OSS and OSS$_L$), although we can see more significant throughput improvement, battery saving is not significant because of the overhead (recall the default ramp/tail time is 8 secs). This highlights the importance of batching and the advantage of the BOSS$_L$ algorithm.



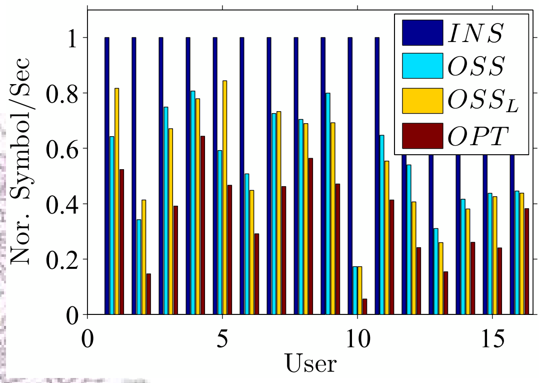
(a)



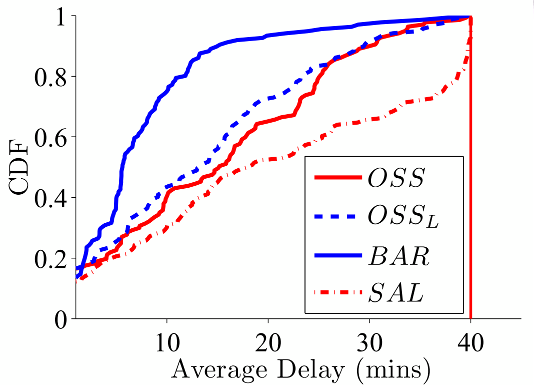
(a)



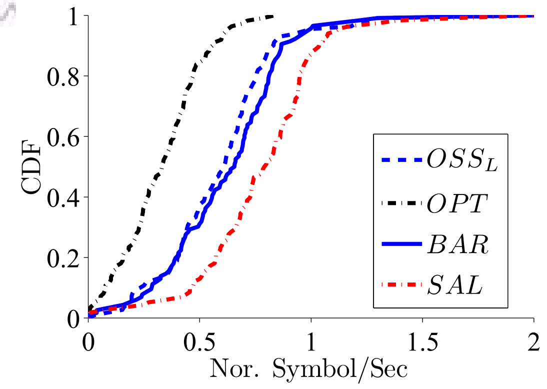
(b)



(b)



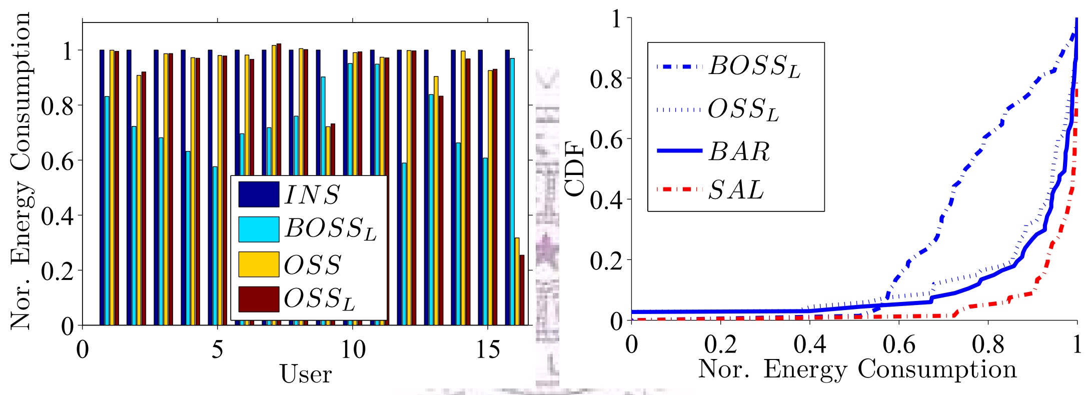
(c)



(c)

Figure 3.5: Throughput improvement: (a) sample normalized throughput, (b) overall throughput improvement, and (c) average delay.

Figure 3.6: Network load reduction: (a) sample network load, (b) sample normalized network load, and (c) overall normalized network load.



(a) (b)

Figure 3.7: Energy consumption reduction: (a) sample normalized energy consumption and (b) overall normalized energy consumption.

Chapter 4

Quantify and Reduce Training Overhead

In this chapter, we study the scheduling model training overhead and we also study how to reduce the overhead without too much performance loss.

4.1 Training Window Size

In Sec. 3.4, we evaluate our proposed scheduling schemes through trace-driven simulations, where the complete profiles are known to the simulator. However, in a real system, the user profiles are accumulated daily. In this chapter, we consider a more practical system, in which the scheduling schemes are trained with increasingly longer profiles.

In particular, we consider a system that works as follows. First, each mobile device uploads profiles to the server every τ days, and the t-th upload consists of the profile collected during $[(t-1)\tau, t\tau)$. This is denoted as *profile upload* in Fig. 4.1. Second, the mobile device downloads the model parameters derived from the profiles collected during training window $[(t-L)\tau, (t-1)\tau)$, where the *training windows size* L is a system parameter. More precisely, the model parameters were derived after preceding synchronization time, using L days of profiles. This step is denoted as *model parameter download* in the figure. Third, the server starts to derive the model parameters using the profiles collected during $[(t-L+1)\tau, t\tau)$. As illustrated in the figure, *model parameter derivation* may take some time in the background, and the resulting model parameters will be downloaded at the next profile synchronization time.

While our system provides the freedom of choosing different training window size L. Selecting the best L is no easy task, as short profiles may lead to unreliable model parameters, but long profiles may incur too much noise. We set up a series of rigorous evaluations. In the simulations, we adopt the Photosync traces from 1041 users. Among

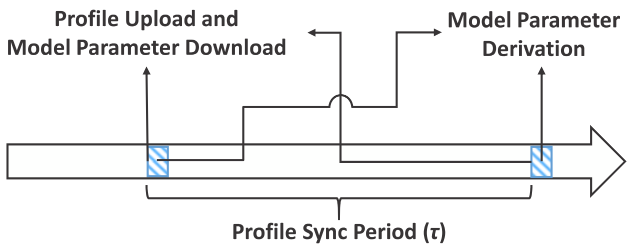


Figure 4.1: System synchronization time line.

the users, we select 10 sample users with the longest profiles, which represent 1282 days of traces in total. We modify the simulator developed in Sec. 3.4 and run simulations with different L values. The simulation results are shown in following.

4.2 Limitations of OSS

We observe that OSS has the best performance with 15-day training window when optimized for throughput. Moreover, the results show that OSS_L outperforms OSS in most cases. In particular, OSS_L outperforms INS by 5.6 times, while OSS outperforms INS by 3.6 times on average. In addition, OSS consumes more resources compared with OSS_L when training the model parameters: OSS consumes up to 3900 times longer training time and up to 450 times larger memory. Table 4.1 summarizes the results from various algorithms and optimization criteria. This table clearly shows that OSS leads to inferior performance, while consumes much more resources. Hence, we do not discuss OSS in the rest of this paper.

4.3 Implication of Training Window Size

In this section, we study the impact of training window size and determine the best training window size via traces-driven simulations.

4.3.1 Single-Job Scheduling

In Fig. 4.2, we report the results of OSS_L algorithm with different L values when optimized for throughput and network load. In Fig. 4.2(a), we show the average throughput normalized to INS algorithm and we also show the average model training time when optimized for throughput. We make three observations from the results. First, the throughput and model training time of OSS_L increases with longer L. Second, when L exceeds 30 days, the model training time continuously increases but the throughput decreases. Last, OSS_L achieves the best throughput when L is between 15 and 30 days, and thus we rec-

Table 4.1: Performance and overhead of OSS, $\mathrm{OSS_L}$ and $\mathrm{BOSS_L}$ with optimal L for OSS

Opt. for Throughput with $L = 15$ days					
	Nor. Performance			**Training Cost**	
Algo.	**Min**	**Mean**	**Max**	**Time (sec)**	**Memory**
OSS	0.01	3.56	5.55	238.67	3.96 GB
$\mathrm{OSS_L}$	0.01	5.55	14.16	0.06	9 KB
Opt. for Network Load with $L = 60$ days					
OSS	0.001	0.56	2.73	231.46	3.96 GB
$\mathrm{OSS_L}$	0.001	0.65	22.98	0.07	9 KB
Opt. for Energy with $L = 30$ days					
OSS	0.003	0.93	4.5	278.82	3.96 GB
$\mathrm{OSS_L}$	0.003	0.90	3.08	0.08	9 KB
$\mathrm{BOSS_L}$	0.001	0.47	3.28	38.96	576 KB

ommend $L \in [15, 30]$ days for $\mathrm{OSS_L}$ when optimized for throughput. In Fig. 4.2(b), we show the results normalized to INS algorithm when optimized for network load. We find that the network load decreases and the model training time increases with longer L. We find the performance of $\mathrm{OSS_L}$ with 15 and 60 days training window are similar and $\mathrm{OSS_L}$ has shorter model training time when L is between 30 and 45 days. Hence, we recommend $L \in [30, 45]$ days for $\mathrm{OSS_L}$ algorithm when optimized for network load.

4.3.2 Multiple-Job Scheduling

We also show the results with batching when optimized for energy consumption in Fig. 4.3. Fig. 4.3(a) shows that with OSS_L the energy consumption decreases and the model training time increases with longer L. For a good tradeoff between performance and overhead, we recommend $L \in [30, 45]$ days for OSS_L when optimized for energy consumption. Fig. 4.3(b) shows that, with BOSS_L, the energy consumption decreases with longer L. However, different to the results from OSS_L, the model training time does not increase with longer L. We recommend $L \in [30, 60]$ for $\mathrm{BOSS_L}$ algorithm when optimized for energy consumption.

In summary, we found that: (i) OSS incurs high training overhead but results in inferior performance and (ii) $\mathrm{OSS_L}$ and $\mathrm{BOSS_L}$ work the best with $L = 30$, in general. We let $L = 30$ in the rest of this article if not otherwise specified.

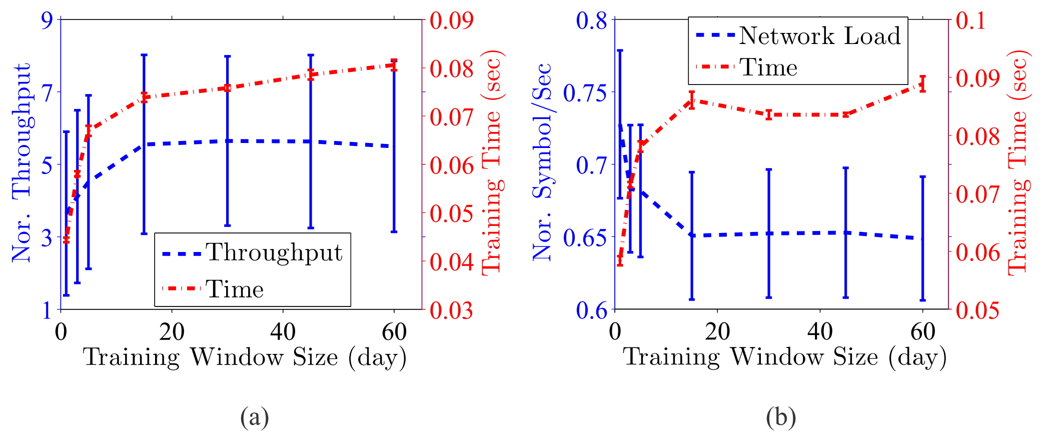


Figure 4.2: Performance of $\mathrm{OSS_L}$ with different training window size when optimized for (a) throughput and (b) network load.

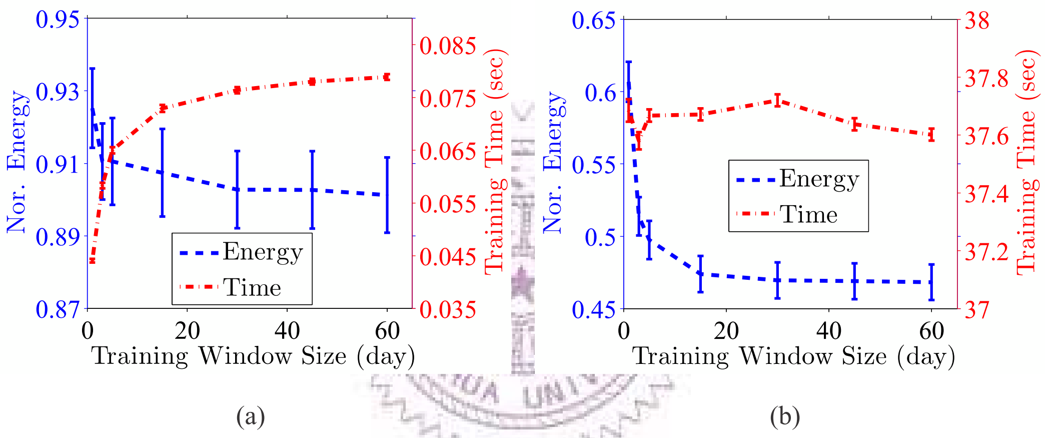


Figure 4.3: Performance of (a) $\mathrm{OSS_L}$ and (b) $\mathrm{BOSS_L}$ with different training window size when optimized for energy consumption.

4.4 Reducing Model Derivation Overhead

We had mentioned the two kinds of overhead when deriving the model parameters. First, the server responsible for generating model parameters may be overloaded when the number of users is large. Second, it takes a month for new users of our system to achieve the optimal performance. In this section, we introduce our proposed method to reduce model parameters derivation overhead and evaluate our method via traces-driven simulations.

4.4.1 User Clustering

To mitigate the overhead, we propose to cluster users with similar behaviors in the same group. Users in the same group share the model parameters. That is, we *combine* their profiles and train a *single* set of model parameters for each group. We consider different user behaviors, referred to as *contexts*, when clustering users. There are many clustering techniques proposed in the data mining literature [8]. Our goal is to cluster mobile users without scarifying the prediction accuracy too much. We add various well-known clustering algorithms in our simulator, which are detailed below.

The updated system works as follows. First, our system runs user clustering algorithms on the user profiles of 1041 PhotoSync users before the model parameters derivation. Second, our system trains the model parameters for each cluster according to the user clustering results. We consider three different clustering algorithms: K-Means, K-Medoids, and hierarchical clustering algorithms, and four different distances: euclidean, cityblock, cosine and correlation distances [8]. To adopt the clustering algorithms, we use the optimization criterion as the context when cluster users and we partition the context into T-sec timeslots with a vector representation. The vector representation contains the average value of the context, i.e., throughput in each timeslots,. In order to show the changing of the context in each timeslot, the context in our vector representation is normalized to a range of $[0, 1]$. The three clustering algorithms have an input parameter: the number of clusters K. To empirically determine the best K, we define clustering ratio α as the ratio between K and number of users N, i.e., $\alpha = \frac{K}{N}$.

4.4.2 Impact of System Parameters

To find the best system parameters: T and α, we design experiments to study the impact of different T and α. In the experiments, we employ hierarchical clustering algorithm and cosine distance for clustering users. In Fig. 4.4, we show the performance and models training time saving of OSS_L algorithm with user clustering when optimized for throughput, network load and energy consumption with different timeslot size and $\alpha = 0.3$. In the figure, the performance and training time are normalized to the results of OSS_L without clustering. We make two observations: (i) the timeslot size do not impact on the models training time saving and (ii) the best T when optimized for throughput, network load and energy consumption are 900-sec, 1800-sec and 600-sec. Based on the results, we consider the best T in the rest of our experiments. In Fig. 4.5, we show results when optimized for throughput, network load and energy consumption with different α and the best T for each optimization criteria. The simulations reveal that mobile user clustering allows us to reduce the overhead of training prediction models yet achieving reasonable performance. For example. OSS_L with clustering can achieve 92.6% of original throughput, 5.2% additional network load and 25.5% additional energy with only 30% of original model parameters training time when $\alpha = 0.3$. Based on the results, we consider $\alpha = 0.3$ in the rest of our experiments.

Table 4.2: The complexity of clustering algorithms

Average Running Time (sec)			
Distance	**Hierarchical**	**K-Means**	**K-Medoids**
Cosine	0.01	1.27	15.80
Euclidean	0.01	0.53	2.05
Cityblock	0.01	0.32	1.97
Correlation	0.01	0.86	2.13

4.4.3 Performance Impact

We first report the average running time of clustering algorithms in Table 4.2 and make two observations. First, K-Medoids algorithm consumes the longest time and hierarchical clustering is the fastest algorithm. Second, cosine distance consumes the longest time, while cityblock distance consumes the shortest time when clustering users.

Next, we report the *performance ratio*, which describes the performance loss due to clustering Fig. 4.6. In particular, this figure reports the performance ratio with and without clustering, when optimized for throughput, network load, and energy consumption. Fig. 4.6(a) shows that: (i) clustering may even lead to higher throughput and (ii) K-Means algorithm with cityblock distance achieves the highest throughput improvement: about 12%. Fig. 4.6(b) shows that K-Medoids algorithm with cityblock distance results in the smallest performance ratio of 118%. Fig. 4.6(c) shows that K-Means algorithm with cosine distance leads to the smallest performance ratio of 117%. In summary, we recommend users to use: (i) K-Means/cityblock, (ii) K-Medoids/cityblock, and (iii) K-Means/cosine when optimized for: (i) throughput, (ii) network load, and (iii) energy consumption, respectively.

4.4.4 Reducing Time Overhead

Last, we report the total running time saving of the clustering algorithms and model parameter derivation. We only give the numbers for the recommended clustering algorithms and distances. We found that the total running time savings of our user clustering algorithms are 58.8%, 37.5% and 59.9% for optimizing for throughput, network load, and energy consumption, respectively. Notice that, different to the time saving when we discuss the impact of the system parameters: α and T, the time consumption here includes the clustering algorithm running time. This demonstrates the effectiveness of our overhead reduction approach.

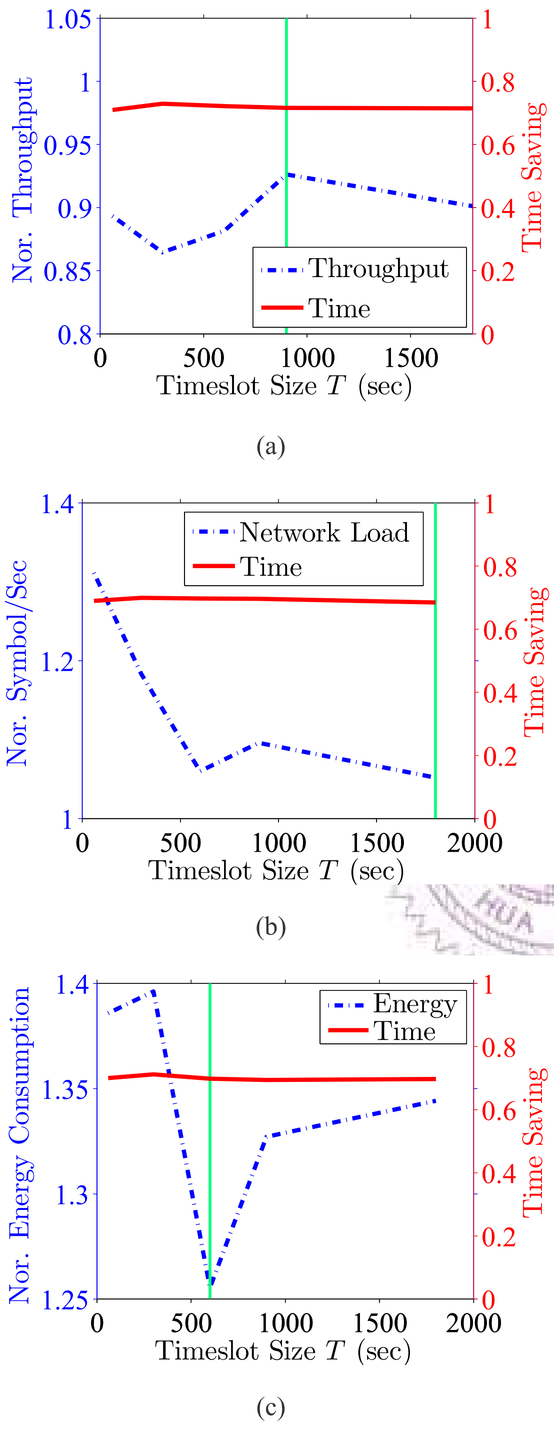


Figure 4.4: Performance and models training time saving when optimized for (a) throughput (b) network load and (c) energy consumption with different clustering timeslot size T.

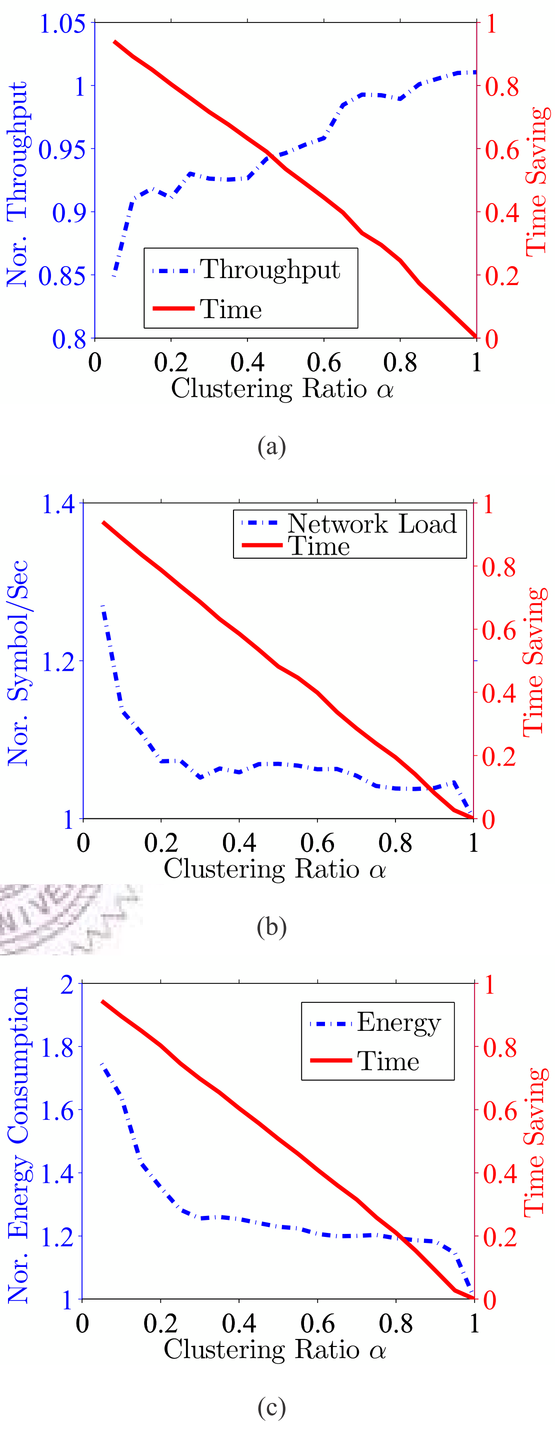
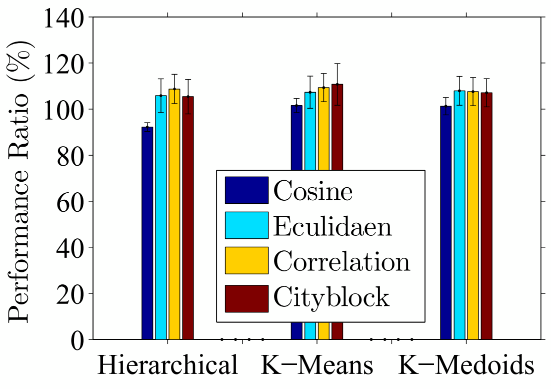
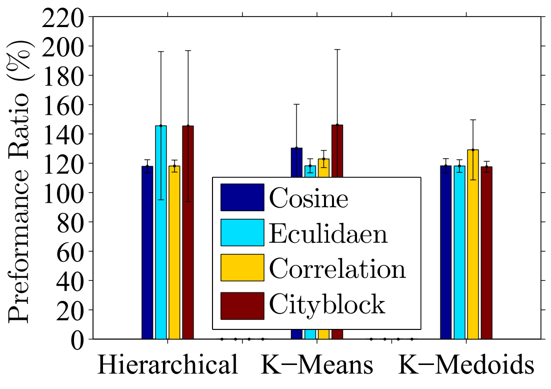


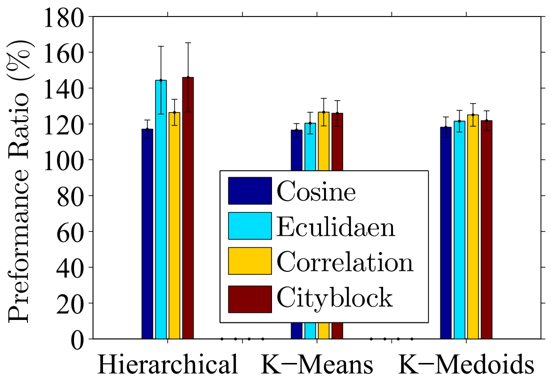
Figure 4.5: Tradeoff between performance and models training time saving when optimized for (a) throughput (b) network load and (c) energy consumption with different clustering ratio α.



(a)



(b)



(c)

Figure 4.6: Performance ratio of user clustering when optimized for (a) throughput, (b) network load, and (c) energy consumption.

Chapter 5

Conclusion and Future Work

In this paper, we propose and implement UPDATE, an user-profile-driven opportunistic data transfer scheduling framework for improved battery performance and network efficiency. Different to existing work, we focus on large time-scale scheduling with deadline guarantee. Large time-scale provides more room for performance improvement, and the deadline guarantee offers predictable user experience and thus eases user adoption.

We evaluate the performance of our proposed UPDATE framework based on real traces from the general public. It shows that a majority of the mobile users can benefit from the UPDATE framework. We also study the overhead of training the model parameters. In order to reduce the training overhead, we propose to cluster users and train single set of model parameters in each group. We also evaluate the reduction of training overhead and the performance degradation with user clustering.

Future work. Our work focuses on large-time scale. When significant network changes occur in small time scale, e.g., when walking or driving, it is desirable to carefully study realistic implementations that span both small and large time scales in an adaptive fashion, with low energy overhead and low computational complexity. We also plan to consider a hybrid optimization objective in our scheduling e.g., consider network loading information from network provider when optimized for throughput to archive better performance. In our work, we only consider the transfer cost in different time. Last, we plan to consider other contexts, e.g., location to predict transfer cost.

Moreover, we only use optimization criteria as the contexts when clustering users and we did not consider batching when clustering users. We plan to rigorously determine the best context for clustering users and to propose a new clustering approach that incorporates the batched transfers.

Bibliography

[1] Photosync. `https://play.google.com/store/apps/details?id=com.metaisle.photosync`.

[2] User's guide, 66321B/D mobile communications dc source. `http://cp.literature.agilent.com/litweb/pdf/5964-8184.pdf`.

[3] 3GPP TS 36.211. Evolved universal terrestrial radio access (E-UTRA); physical channels and modulation. `http://www.3gpp.org/specifications`.

[4] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3G using WiFi. In *Proc. of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, June 2010.

[5] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *ACM SIGCOMM Conference on Internet Measurement Conference (IMC '09)*, November 2009.

[6] M. Bayir, N. Eagle, and M. Demirbas. Discovering spatiotemporal mobility profiles of cellphone users. In *Proc. of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM '09)*, June 2009.

[7] S. Ha, S. Sen, C. Joe-Wong, Y. Im, and M. Chiang. TUBE: Time-dependent pricing for mobile data. In *Proc. of ACM SIGCOMM'12*, August 2012.

[8] J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2005.

[9] W. Hsu, D. Dutta, and A. Helmy. Structural Analysis of User Association Patterns in University Campus Wireless LANs. *IEEE Transactions on Mobile Computing*, 11(11), 2012.

[10] W. Hsu, T. Spyropoulos, K. Psounis, and A. Helmy. Modeling timevariant user mobility in wireless mobile networks. In *Proc. of IEEE International Conference on Computer Communications (INFOCOM '07)*, May 2007.

[11] H. Lu, S. Tseng, and S. Yu. Mining cluster-based temporal mobile sequential patterns in location-based service environments. *IEEE Transactions on Knowledge and Data Engineering*, 23(6), 2011.

[12] H. Ma, H. Cao, Q. Yang, E. Chen, and J. Tian. A habit mining approach for discovering similar mobile users. In *Proc. of ACM International Conference on World Wide Web (WWW '12)*, April 2012.

[13] P. Mohan, S. Nath, and O. Riva. Prefetching mobile ads: can advertising systems afford it? In *Proc. of the ACM European Conference on Computer Systems (EuroSys '13)*, April 2013.

[14] M. Ra, J. Paek, A. Sharma, R. Govindan, M. Krieger, and M. Neely. Energy-delay tradeoffs in smartphone applications. In *Proc. of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, June 2010.

[15] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. Padmanabhan. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proc. of ACM Annual International Conference on Mobile Computing and Networking (MobiCom'10)*, September 2010.

[16] I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci. Taming user-generated content in mobile networks via drop zones. In *Proc. of IEEE International Conference on Computer Communications (Infocom '11)*, April 2011.

[17] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proc. of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, June 2012.

[18] J. Ying, H. Lu, W. Lee, T. Weng, and S. Tseng. Mining user similarity from semantic trajectories. In *Proc. of ACM International Workshop on Location Based Social Networks (LBSN '10)*, November 2010.

[19] Y. Zheng, L. Zhang, X. Xie, and W. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proc. of ACM International Conference on World Wide Web (WWW '09)*, April 2009.