Profiling Resource Usage for Mobile Applications: A Cross-layer Approach

Feng Qian
University of Michigan

Zhaoguang Wang
University of Michigan

Alexandre Gerber
AT&T Labs Research

Z. Morley Mao
University of Michigan

Subhabrata Sen
AT&T Labs Research

Oliver Spatscheck
AT&T Labs Research

ABSTRACT

Despite the popularity of mobile applications, their performance and energy bottlenecks remain hidden due to a lack of visibility into the resource-constrained mobile execution environment with potentially complex interaction with the application behavior. We design and implement *ARO*, the mobile **A**pplication **R**esource **O**ptimizer, the first tool that efficiently and accurately exposes the cross-layer interaction among various layers including radio resource channel state, transport layer, application layer, and the user interaction layer to enable the discovery of inefficient resource usage for smartphone applications. To realize this, ARO provides three key novel analyses: *(i)* accurate inference of lower-layer radio resource control states, *(ii)* quantification of the resource impact of application traffic patterns, and *(iii)* detection of energy and radio resource bottlenecks by jointly analyzing cross-layer information. We have implemented ARO and demonstrated its benefit on several essential categories of popular Android applications to detect radio resource and energy inefficiencies, such as unacceptably high (46%) energy overhead of periodic audience measurements and inefficient content prefetching behavior.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design – Wireless Communication; C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Smartphone Applications, Radio Resource Optimization, Cross-layer Analysis, RRC state machine, UMTS, 3G Networks

1. INTRODUCTION

Increasingly ubiquitous cellular data network coverage gives an enormous impetus to the growth of diverse smartphone applications. Despite a plethora of such mobile applications developed by both the active user community and professional developers, there remain far more challenges associated with mobile applications compared to their desktop counterparts. In particular, application developers are usually unaware of cellular specific characteristics that incur potentially complex interaction with the application behavior. Even for professional developers, they often do not have visibility into the resource-constrained mobile execution environment. Such situations potentially result in smartphone applications that are not cellular-friendly, *i.e.,* their radio channel utilization or device energy consumption are inefficient because of a lack of transparency in the lower-layer protocol behavior. For example, we discovered that for Pandora, a popular music streaming application on smartphones, due to the poor interaction between the radio resource control policy and the application's data transfer scheduling mechanism, 46% of its radio energy is spent on periodic audience measurements that account for only 0.2% of received user data (§7.2.1).

In this work, we address the aforementioned challenge by developing a tool called ARO (mobile **A**pplication **R**esource **O**ptimizer). To the best of our knowledge, ARO is the first tool that exposes the *cross-layer interaction* for layers ranging from higher layers such as user input and application behavior down to the lower protocol layers such as HTTP, transport, and very importantly radio resources. In particular, so far little focus has been placed on the interaction between applications and the radio access network (RAN) in the research community. Such cross-layer information encompassing device-specific and network-specific information helps capture the tradeoffs across important dimensions such as energy efficiency, performance, and functionality, making such tradeoffs explicit rather than arbitrary as it is often the case today. It therefore helps reveal inefficient resource usage (*e.g.,* high resource overhead of periodic audience measurements for Pandora) due to a lack of transparency in the lower-layer protocol behavior, leading to suggestions for improvement.

We choose UMTS (the Universal Mobile Telecommunications System) network, one of the most popular 3G mobile communication technologies [1], as the target RAN for our ARO prototype. In UMTS, the key factor affecting application performance and network energy efficiency is the Radio Resource Control (RRC) state machine [24] whose purpose is to efficiently utilize limited radio resources and to improve handset battery life. A handset (*i.e.,* the mobile device) can be in one of three RRC states (Figures 1 and 2) each with vastly different amount of allocated radio resources and power. Application traffic patterns trigger RRC state transitions, which in turn affect radio resource utilization, handset energy consumption, and user experience. Awareness of the RRC states and their transition behavior is essential to ensuring efficient network energy usage [24, 25].

MobiSys'11, June 28–July 1, 2011, Bethesda, Maryland, USA.

ARO bridges the aforementioned gap, *i.e.,* a lack of visibility of cellular-specific characteristics hinders developers from building cellular-friendly applications. In particular, ARO performs various analyses for RRC layer, TCP layer, HTTP layer, user interactions, followed by their cross-layer interactions, to reveal the impact of smartphone applications on radio resources and battery life.

ARO consists of an online lightweight data collector and an offline analysis module. To profile an application, an ARO user simply starts the data collector, which incurs less than 15% of runtime overhead, and then runs the application for a desired duration as a normal application user. The collector captures packet traces, system and user input events, which are subsequently processed by the analysis module on a commodity PC. The proposed ARO framework (§3) also applies to other types of cellular networks such as GPRS/EDGE, EvDO, and 4G LTE that involve similar tradeoffs to those in UMTS (§2). We highlight our contributions as follows.

1. **An accurate RRC state inference technique (§4).** We present a methodology that accurately infers RRC states from packet traces collected on a handset. The inference technique is necessary due to lacking of an interface for accessing RRC states directly from the handset hardware. Using a simulation-based approach to infer RRC states, the new inference algorithm differs from previous work [24, 25] in two aspects. First, the previous algorithm assumes traces are collected at the cellular core network while our new approach targets at a more common scenario where traces are captured directly on a handset. Second, our algorithm significantly improves the inference accuracy by performing more fine-grained simulation of transmission queues to more precisely capture state transitions. As shown in §4.3, considering such new factors increases the accuracy of state transition identification from 85% to 98%. We devise a novel power-based inference algorithm to validate our packet-based inference approach (§4.3).

2. **Root cause analysis for short traffic bursts (§5.2).** Low efficiency of radio resource and energy usage are fundamentally attributed to *short traffic bursts* carrying small amount of user data while having long idle periods, during which a device keeps the radio channel occupied, injected before and after the bursts [10, 24]. We develop a novel algorithm to identify them and to distinguish which factor triggers each such burst, *e.g.,* user input, TCP loss, or application delay, by synthesizing analysis results of the TCP, HTTP, and user input layer. ARO also employs a robust algorithm (§5.2.1) to identify periodic data transfers that in many cases incur high resource overhead. Discovering such triggering factors is crucial for understanding the root cause of inefficient resource utilization. Previous work [13, 24] also investigate the impact of traffic patterns on radio power management policy and propose suggestions. In contrast, ARO is essential in providing more specific diagnosis by breaking down resource consumption into each burst with its triggering factor accurately inferred. For example, for the Fox News application (§7.2.2), by correlating application-layer behaviors (*e.g.,* transferring image thumbnails), user input (*e.g.,* scrolling the screen), and RRC states, ARO reveals it is user's scrolling behavior that triggers scattered traffic (*i.e.,* short bursts) for downloading image thumbnails in news headlines (*i.e.,* images are transferred only when they are displayed as a user scrolls down the screen), and quantifies its resource impact. Analyzing data collected at one single layer does not provide such insight due to incomplete information (Table 6).

3. **Quantifying resource impact of traffic bursts (§5.3).** In order to *quantitatively* analyze resource bottlenecks, ARO addresses a new problem of quantifying resource consumption of traffic bursts due to a certain triggering factor. It is achieved by computing the difference between the resource consumption in two scenarios where bursts of interest are kept and removed, respectively. The latter scenario requires changing the traffic pattern. To address such a challenge of modifying a cellular packet trace while having its RRC states updated accordingly, ARO strategically decouples the RRC state machine impact from application traffic patterns, modifies the trace, and then faithfully reconstructs the RRC states.

4. **Identification of resource inefficiencies of real Android applications (§7).** We apply ARO to six real Android applications each with at least 250,000 downloads from the Android market as of Dec 2010. ARO reveals that many of these very popular applications (Fox News, Pandora, Mobclix ad platform, BBC News *etc.*) have significant resource utilization inefficiencies that are previously unknown. We provide suggestions on improving them. In particular, we are starting to contact developers of popular applications such as Pandora. The feedback has been encouragingly positive as the provided technique greatly helps developers identify resource usage inefficiencies and improve their applications [2].

The rest of the paper is organized as follows. After providing sufficient technical background in §2, we outline the ARO system in §3. In §4, we detail the RRC state inference technique with its evaluation, followed by analyses at higher layers (TCP, HTTP, burst analysis) as well as the cross-layer synthesis in §5. We briefly describe how we implement the ARO prototype in §6, then present case studies of six Android applications in §7 to demonstrate typical usage of ARO. We summarize related work in §8 before concluding the paper in §9.

2. BACKGROUND

This section provides background for further discussion.

The UMTS Architecture. The UMTS network consists of three subsystems: handsets (mobile devices), UMTS Terrestrial Radio Access Network (UTRAN), and the Core Network (CN) [17].
UTRAN, the radio access network connecting handsets and CN, consists of two components: Node-B (*i.e.,* base stations), and Radio Network Controllers (RNC). RNC is a governing element in UTRAN and is responsible for controlling multiple Node-Bs. The centralized CN is the backbone of the cellular network.

The RRC States. In UMTS networks, the Radio Resource Control (RRC) protocol introduces for each handset a state machine to efficiently utilize the limited radio resources (*i.e.,* WCDMA codes). A single RRC state machine is maintained at both a handset and the RNC. The two peer entities are always synchronized via control channels except during transient and error situations [17]. Typically there are three RRC states: IDLE, CELL_DCH, and CELL_FACH. We refer to them as IDLE, DCH, and FACH thereafter, respectively.

When a handset is turned on, it is at the IDLE state by default. The handset has not established an RRC connection with the RNC, thus no radio resource is allocated and the handset cannot transfer any user data. The power consumption of its radio interface is almost zero at IDLE[1]. At the DCH state, the RRC connection

[1]Some UMTS networks support a hibernating state called CELL_PCH. It is similar to IDLE but the state promotion delay from CELL_PCH is shorter.

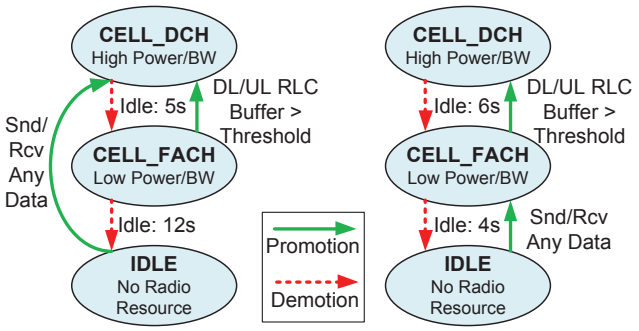

Figure 1: RRC State Machine (Carrier 1)

Figure 2: RRC State Machine (Carrier 2)

is established and a handset is usually allocated *dedicated* transport channels[2] for both downlink (DL, RNC→handset), and uplink (UL, handset→RNC), allowing the handset to fully utilize radio resources for high-speed user data transmission. The radio power consumption at DCH is the highest (600 to 800 mW). The FACH state is an intermediate state between IDLE and DCH. At FACH, the RRC connection is established but there are only low-speed *shared* channels (for both downlink and uplink) allocated to a handset. FACH is designed for applications with low data throughput rate. Radio power at FACH is 55% to 75% of that at DCH (§4.4).

State Transitions. There are two types of RRC state transitions: promotions (IDLE→DCH, FACH→DCH, and IDLE→FACH), and demotions (going in reverse directions). Promotions (demotions) switch from a state with lower (higher) radio resource and radio power consumption to another state requiring more (less) radio resources and power. Figure 1 and Figure 2 show the RRC state machines of two large U.S. commercial UMTS carriers (denoted as Carrier 1 and 2, respectively) whose state transition models are inferred by our previous work [24]. As illustrated, promotions are triggered by user data transmission. A promotion from IDLE begins when *any* user data transmission activity happens, while a FACH→DCH promotion takes place when a handset sends or receives data at a higher speed, *i.e.,* the per-device queue size, called Radio Link Controller (RLC) buffer size, exceeds a threshold in either direction (uplink and downlink use separate RLC buffers). On the other hand, state demotions are triggered by two inactivity timers configured by the RNC. We denote the DCH→FACH timer as α, and the FACH→IDLE timers as β. At DCH state, the RNC resets the α timer to a constant threshold T whenever it observes any data frame. If there is no user data transmission for T seconds, the α timer expires and the state is demoted to FACH. The β timer uses a similar scheme.

Promotion Delays and Tail Times distinguish cellular networks from other types of access networks. An RRC state promotion incurs a long latency (up to 3 seconds) during which tens of control messages are exchanged between a handset and the RNC for resource allocation. A large number of state promotions incur high signaling overhead as they increase processing load at the RNC and worsen user experience [8]. In contrast, state demotions finish much faster, but they incur *tail times* that cause significant waste of resources [10, 24, 25]. A *tail* is the idle time period matching the inactivity timer value before a state demotion. During a tail time, a handset simply waits for the inactivity timer to expire, but it still occupies transmission channels and WCDMA codes, and its radio power consumption is kept at the corresponding level of the state. Due to the tail time, transmitting even small amount of data can cause significant radio energy and radio resource consumption.

[2]A UE can access HSDPA/HSUPA (High Speed Downlink/Uplink Packet Access) mode, if supported by the infrastructure, at DCH state. For HSDPA, the high speed transport channel is not dedicated, but shared by a limited number (*e.g.,* 32) of users [17].

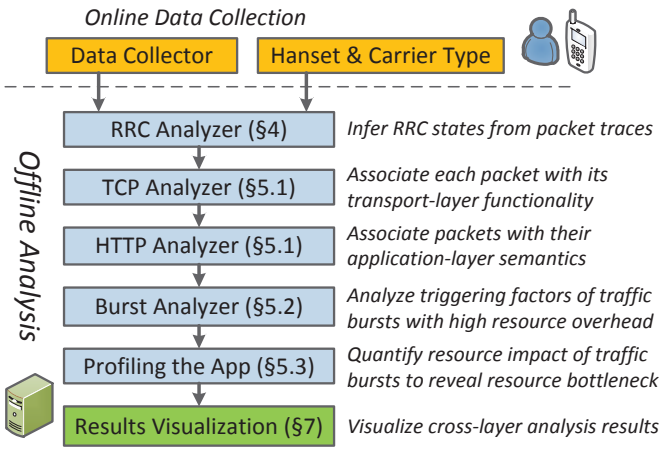

Figure 3: The ARO System

Other Types of Radio Access Networks. Promotion delays, tail times, and their associated tradeoffs [24], also exist in other types of radio access networks (*e.g.,* GPRS/EDGE [6] and EvDO [12]). A similar battery life versus latency tradeoff exists in 4G LTE network due to its DRX (Discontinuous Reception) mechanism, which involves a state machine similar to the one used by UMTS [27]. The ARO framework (§3) applies to all the above types of networks.

3. ARO OVERVIEW

This section outlines the ARO system, which consists of two main components: the data collector and the analyzers. The data collector runs efficiently on a handset to capture information essential for understanding resource usage, user activity, and application performance. Our current implementation collects network packet traces and user input events. But other information such as application activities (*e.g.,* API calls) and system information (*e.g.,* CPU usage) can also be collected for more fine-grained analysis. The collected traces are subsequently fed into the analyzers, which run on a PC, for offline analysis. Our design focuses on modularity to enable independent analysis of individual layers whose results can be subsequently correlated for joint cross-layer analysis. The proposed framework is easily extensible to other analyzers of new application protocols. We describe the workflow of ARO as outlined in Figure 3.

1. The ARO user invokes on her handset the data collector, which subsequently collects relevant data, *i.e.,* all packets in both directions and user input (*e.g.,* tapping or scrolling the screen). Unlike other smartphone data collection efforts [28, 14], our ability to collect user interaction events and packet-level traces enables us to perform fine-grained correlation across layers. ARO also identifies the packet-to-application correspondence. This information is used to distinguish the *target application*, *i.e.,* the application to be profiled, from other applications simultaneously accessing the network. Note that ARO collects all packets since RRC state transitions are determined by the aggregated traffic of all applications running on a handset.

2. The ARO user launches the target application and uses the application as an end user. Factors such as user behavior randomness and radio link quality influence the collected data and thus the analysis results. Therefore, to obtain a representative understanding of the application studied, ARO can be used across multiple runs or by multiple users to obtain a comprehensive exploration of different usage scenarios of the target application, as exemplified in our case studies (§7.1). The target application might also be explored in several usage scenarios, covering diverse functionalities, as well as execution modes (*e.g.,* foreground and background).

3. The ARO user loads the ARO analysis component with the collected traces. ARO then configures the RRC analyzer with handset and carrier specific parameters, which influence the model used for RRC analysis (§4). The TCP, HTTP, and burst analyzers are generally applicable.

4. ARO then performs a series of analyses across several layers. In particular, the RRC state machine analysis (§4) accurately infers the RRC states from packet traces so that ARO has a complete view of radio resource and radio energy utilization during the entire data collection period. ARO also performs transport protocol and application protocol analysis (§5.1) to associate each packet with its transport-layer functionality (*e.g.,* TCP retransmission) and its application-layer semantics (*e.g.,* an HTTP request). Our main focus is on TCP and HTTP, as the vast majority of smartphone applications use HTTP over TCP to transfer application-layer data [21, 16]. ARO next performs burst analysis (§5.2), which utilizes aforementioned cross-layer analysis results, to understand the triggering factor of each short traffic burst, which is the key reason of low efficiency of resource utilization [24].

5. ARO profiles the application by computing for each burst (with its inferred triggering factor) its radio resource and radio energy consumption (§5.3) in order to identify and quantify the resource bottleneck for the application of interest. Finally, ARO summarizes and visualizes the results. Visualizing cross-layer correlation results helps understand the time series of bursts that are triggered due to different reasons, as later demonstrated in our case studies (§7).

4. INFERRING RRC STATES

We present our algorithm to accurately infer RRC states and state transitions from packet traces collected on handsets. We focus on describing the inference algorithm for Carrier 1 (Figure 1) while the technique is also applicable to other carriers using a different state machine through minor modification.

Clearly, this problem can be cleanly solved if the online data collector is able to read the RRC state from the handset hardware. However, we are not aware of any API or known workaround for directly accessing the RRC state information on any smartphone system. In other words, it is difficult to directly observe the low-level communication between a handset and the RNC.

Previous effort [24, 25] also applies a simulation-based approach to estimate RRC state machine statistics. Our approach differs from previous work in two aspects. First, the previous algorithm assumes traces are collected at the cellular core network while our approach targets at a more common scenario where traces are captured directly on a handset. Second, our algorithm significantly improves the inference accuracy by performing more fine-grained simulation of uplink and downlink RLC buffers to more precisely capture state promotions.

4.1 Measuring State Machine Parameters

We first conduct controlled experiments to measure Carrier 1's state machine parameters. To our knowledge, this is to date the most comprehensive characterization of the RRC state machine behavior for a commercial cellular carrier in terms of measured parameters. These parameters shown in Table 1 are statically configured by the RNC while different RNCs may adopt different values[3]. Therefore prior to use, the ARO analyzer needs to be configured with (automatically inferred) local RRC parameters, which are used by the state inference algorithm in §4.2.

4.1.1 Basic State Machine Parameters

We first describe three basic state machine parameters.

Inactivity timers control the release of radio resources. Using our previously proposed methodology [24], we infer the DCH→FACH and the FACH→IDLE timers to be 5 sec and 12 sec, respectively, for both an HTC TyTN II phone and a Sierra 3G Air card. The timer values are validated by measuring the real-time power of an HTC TyTN II phone (similar to Figure 7).

State Promotion Delay refers to the latency caused by a state promotion. Using the methodology described in previous work [24], we derive the IDLE→DCH promotion delay and the FACH→DCH delay to be 2.0±1.0 sec and 1.5±0.5 sec, respectively. The delays are not constant because the control channel rate, and the workload of the RNC, which processes state promotions, may vary[4].

RLC Buffer Thresholds are essential in determining the promotions from FACH to DCH, as described in §2. We measure the RLC buffer thresholds for uplink (UL) and downlink (DL) separately using the method described in [24]. The measurement results are summarized in Figure 4 where the Y axis corresponds to the probability of observing a FACH→DCH promotion when a packet of x bytes is sent. We observe that for DL, the threshold is fixed to 475 bytes while the RLC buffer threshold for UL varies from 500 to 560 bytes. Such a difference is likely due to the disparity between the UL and DL transport channels used by the FACH state [22].

4.1.2 New State Machine Parameters

We next discuss new state machine parameters that have not been explored by previous work. Considering these new factors in the simulation algorithm increases the accuracy of state promotion identification from 85% to 98% (§4.3).

RLC Buffer Consumption Time quantifies how fast the RLC buffer is cleared after it is filled with data at FACH state. It depends on channel throughput at FACH since the RLC buffer is not emptied until all data in the buffer are transmitted [17]. Considering RLC buffer consumption time enables the state inference algorithm (§4.2) to perform more fine-grained simulation of RLC buffer dynamics to more precisely capture state promotions, thus improving the inference accuracy.

We infer the RLC buffer consumption time by sending two packets separated by some delay. First, we send a packet of x bytes at FACH with x smaller than the RLC buffer threshold so it never

[3]For example, for Carrier 1, RNCs in some big cities employ a DCH→FACH timer of 3 sec instead of the common value of 5 sec.

[4]An IDLE→DCH/FACH promotion triggered by a downlink packet is usually longer than that triggered by an uplink packet. This is because when a downlink packet is to be received, it may get delayed due to *paging*. In fact, even at IDLE, a handset periodically wakes up to listen for incoming packets on the paging channel. If a downlink packet happens to arrive between two paging occasions, it will be delayed until the next paging occasion. In practice, we observe via power monitor that the paging cycle length is 2.56 sec for Carrier 1 and 1.28 sec for Carrier 2.

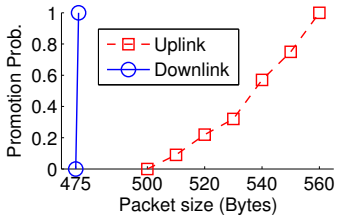

Figure 4: RLC buffer thresholds (UL/DL)

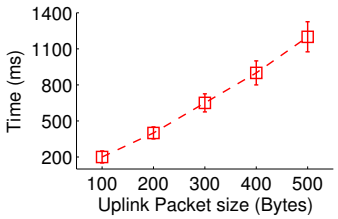

Figure 5: RLC buffer consumption time (UL)

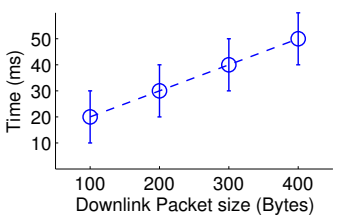

Figure 6: RLC buffer consumption time (DL)

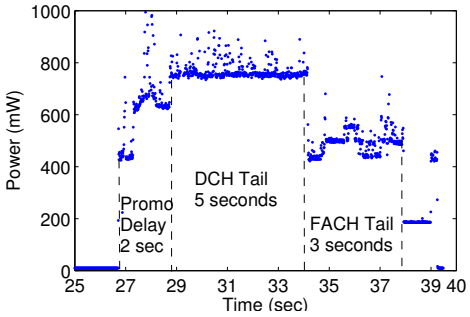

Figure 7: Inactivity timers (tail times) of a Nexus One phone. A single UDP packet is sent at $t = 26.8$ sec.

Table 1: Carrier 1's parameters used for RRC state inference

	TyTn	Sierra	NexusOne
FACH→IDLE Timer (α)	12 seconds		12 sec / 3 sec$^+$
DCH→FACH Timer (β)	5 seconds		
RLC Buffer Thresholds	UL:540 Bytes DL:475 Bytes		
RLC Consumption Time (UL)*	$0.0014x^2 + 1.6x + 20$ msec		
RLC Consumption Time (DL)*	$0.1x + 10$ msec		
IDLE→DCH Delay$^\triangle$	2.0 ± 1.0 sec		
FACH→DCH Delay	1.5 ± 0.5 sec		

$^+$ Depending on the fast dormancy behavior.
* Quadratic curve fitting based on Figure 5 and Figure 6.
$^\triangle$ An IDLE→DCH promotion triggered by a downlink packet can take up to 5 seconds due to paging (see the footnote in §4.1.1). But usually promotions from IDLE are triggered by uplink packets.

triggers a FACH→DCH promotion. After a delay for y milliseconds, another packet of z bytes is sent in the same direction. We fix z at 500 bytes and 475 bytes for uplink and downlink, respectively, according to Figure 4 (the exact value of z does not matter as long as $z <$ the RLC buffer threshold $< x + z$). Then observing a FACH→DCH promotion suggests that the RLC buffer is not yet emptied when the second packet arrives at the buffer, causing the RLC buffer size to exceed the threshold.

Figure 5 shows the RLC buffer consumption time for uplink. For each packet size x (X axis), we vary the delay y (Y axis) at a granularity of 25 ms, and perform aforementioned test for each pair of (x, y) for 20 times. The error bars in Figure 5 cover a range of delays (y values) for which we probabilistically observe a promotion. The results for downlink are shown in Figure 6. Our results confirm previous measurements that at FACH, the uplink transport channel is much slower than the downlink channel [22].

Low traffic volume not triggering timers to reset. We found that for Carrier 1, the DCH→FACH timer is not reset when a handset has very little data to transfer for both directions. Specifically, at DCH, a packet P does not reset the timer if both uplink and downlink have transferred no more than 320 bytes (including P) within the past 300 ms. We believe the intent of such a design, which is specific to Carrier 1 and is not documented by literature [17, 18, 22], is to save radio resources in DCH when there is small traffic demand by a handset. Not considering this factor leads to overestimation of DCH occupation time.

Fast Dormancy is a recently proposed feature in 3GPP specifications [7]. Instead of waiting for inactivity timers to expire, a handset can actively request for a state demotion to IDLE (or to the hibernating CELL_PCH state with a lower promotion delay) by sending a special RRC control message to the RNC. We investigated four handsets using Carrier 1's UMTS network: HTC TyTN II, Sierra 3G Air card, and two Google Nexus One phones (A and B). For TyTn II, the air card, and Nexus One A, their state demotions are solely controlled by inactivity timers. For Nexus One B, the measured α and β timers are 5 sec and only 3 sec (shorter than the default 12-sec β timer), respectively. Such an observation is further validated by measuring the power of Nexus One B (Figure 7). It is highly likely that it employs fast dormancy to release radio resources earlier to improve its battery life. Also we believe that fast dormancy is controlled by the upgradable radio image that distinguishes the two Nexus One phones. The incurred drawbacks of fast dormancy are extra state promotions causing additional signaling overhead and potentially worsening user experience [8, 24].

4.2 RRC State Inference Algorithm

We now describe our state inference algorithm, which takes a packet trace $P_1, ..., P_n$ as input where P_i is the i-th packet in a trace collected on a handset. The output is $S(t)$ denoting the RRC state or state transition at any given time t. $S(t)$ corresponds to one of the following: IDLE, FACH, DCH, IDLE→FACH, and FACH→DCH. We assume the RRC state machine, which can be inferred by other work [24], and its parameters (measured in §4.1) are known.

Our state inference algorithm follows a high-level idea similar to that in [24] by replaying the packet trace against an RRC state machine simulator. The major differences are the following. First, as mentioned before, our algorithm targets at a more common scenario where traces are captured directly on a handset. Second, by considering RLC buffer consumption time (§4.1.2), our algorithm performs more fine-grained simulation of RLC buffer dynamics (both uplink and downlink) to more precisely capture state promotions. Sometimes a FACH→DCH promotion is triggered by multiple small packets that incrementally fill up the RLC buffer, instead of a single large packet with its size exceeding the RLC buffer threshold. Previous approach [24] may miss such promotions. Third, the improved algorithm considers cases where DCH timers are not reset and the handset fast dormancy behaviors. We show the accuracy improvement of our new technique in §4.3.

The algorithm performs iterative packet-driven simulation. Let P_i and P_{i+1} be two consecutive packets whose arrival time are t_i and t_{i+1}, respectively. Intuitively, if $S(t_i)$ is known, then $\forall t_i < t \leq t_{i+1}$, $S(t)$ can be inferred in $O(1)$ based on three factors,

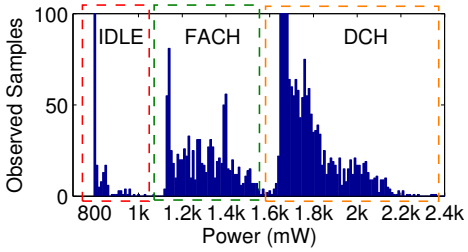

Figure 8: Histogram of measured handset power values for the `News1` trace collected on an HTC TyTn II phone

by following the RRC state transition rules. *(i)* The inter-arrival time between P_i and P_{i+1}, depending on which a handset may experience tail times causing a state demotion then a possible state promotion when P_{i+1} arrives. *(ii)* The packet size of P_{i+1}, which may trigger a FACH→DCH promotion if it fills up the RLC buffer. *(iii)* The direction of P_{i+1}. Assuming that P_{i+1} triggers a promotion. If P_{i+1} is downlink, the promotion already finishes when the handset receives it. Therefore P_{i+1} triggers a promotion *before* its arrival. On the other hand, if P_{i+1} is uplink *i.e.,* the application just puts the packet into the uplink RLC buffer, then the state promotion has just begun. Therefore the promotion will happen *after* P_{i+1} is captured. When $S(t_{i+1})$ is determined, $S(t_{i+2})$ can be iteratively computed based on $S(t_{i+1})$ and P_{i+2}, and so on.

4.3 Validation of State Inference

We evaluate our *simulation-based* state inference technique by comparing it with *handset-power-based* inference approach. We simultaneously collect both power traces (using a hardware power monitor [5]) and packet traces for popular web sites from an HTC TyTn II smartphone using Carrier 1's UMTS network. We infer the RRC states independently from each trace, and then compare their results. Note that ARO itself does not require any special monitoring equipment since it employs simulation-based state inference technique.

4.3.1 Power-based State Inference

Inferring RRC states from power traces requires special monitoring equipment and is non-trivial due to noise. We next describe a novel algorithm that infers RRC states from a handset's *overall* power consumption, since it is difficult to measure the radio interface power separately. Our basic assumptions are *(i)* a handset's 3G radio interface consumes a considerable fraction of the total handset power [31], and *(ii)* the power consumed at the three RRC states differs significantly (§4.4). Both assumptions are confirmed by our experiments (described later in Figure 8).

The input generated by the power meter is $P(t)$ describing the overall handset power at a granularity of 0.2 msec. Our power-based state inference algorithm distinguishes RRC states using fixed power thresholds and identifies state transitions by observing power changes. It consists of three steps. *(i)* Downsample $P(t)$ from 5kHz to 10Hz by averaging power values of every 500 power samples to reduce noise. *(ii)* Use two power thresholds μ and ν to distinguish the three RRC states: $P(t) < \mu$ for IDLE, $\mu \leq P(t) < \nu$ for FACH, and $\nu \leq P(t)$ for DCH. *(iii)* Identify state transitions by examining power changes crossing the thresholds.

The values of μ and ν are determined from histograms of measured power values of each trace. A representative example for an HTC TyTn II phone is shown in Figure 8, from which we observe that the measured overall handset power values form three clusters corresponding to IDLE, FACH, and DCH. The variation within each cluster is mostly due to power change of other system components (*e.g.,* CPU). Figure 8 indicates that the 3G radio interface plays a vital role in determining the overall handset power consumption, as at DCH, the radio power (800 mW) contributes 1/3 to 1/2 of the total device power (1600 mW to 2400 mW[5]). All histograms for traces shown in Table 2 suggest that we set μ and ν to 1000 mW and 1600 mW, respectively, for TyTn II.

Table 2: Packet-based vs. power-based inference results

Trace name (Trace Length)	% Time Overlap	State Promotions		
		Agree	Pkt Err	Pwr Err
`News1` (180s)	93.4%	7	1 (1)	0
`News2` (180s)	96.5%	9	0 (2)	0
`News3` (190s)	96.3%	9	0 (1)	0
`News4` (250s)	95.5%	12	0 (2)	1
`Social1`(250s)	95.7%	13	0 (2)	0
`Social2`(180s)	91.3%	10	0 (2)	0
`Social3`(275s)	94.5%	18	1 (3)	0
`Email1` (250s)	94.4%	16	0 (2)	1
`Email2` (275s)	94.4%	17	0 (3)	0
`Stream1`(180s)	98.7%	3	0 (0)	0
`Stream2`(180s)	99.1%	2	0 (0)	0
Total (2390s)	95.3%	116	2 (18)	2

4.3.2 Validation Results

We obtained 11 traces listed in Table 2 by simultaneously collecting both packet traces and power traces. Then we employ both algorithms (packet-based and power-based) to infer the RRC states. Table 2 shows the comparison results. The "% Time Overlap" column computes the percentage of time periods during which both algorithms produce exactly the same RRC state or the same state promotion.

The right three columns of Table 2 compares inferred state promotions. "Agree" counts the number of pairs (X, Y), where promotion X and Y are inferred by the two methods respectively, such that X and Y have the same promotion type and their time periods overlap (it does not require that they match exactly). For a promotion inferred by either algorithm not belonging to such a pair, it is either an error of packet-based inference methodology (counted by the first number of "Pkt Err") or an error of the power-based approach (counted by "Pwr Err") due to noise, judged by our manual inspection. Table 2 indicates that both inference methods have comparably high accuracy. We observe that inaccuracies are mostly caused by noises for power-based approach, and variations of uplink RLC buffer thresholds and consumption time, for packet-based inference method.

Comparing with a previous inference algorithm. The "Pkt Err" column compares our algorithm (the first number) with a previous simulation approach [24, 25] where RLC buffer consumption rate and cases where the α timer is not reset (§4.1) were not considered (the second number in parenthesis). The results clearly show that performing more fine-grained simulation improves the inference accuracy of state promotion identification from 85% to 98% (*i.e.,* reducing the total errors from 17 to 2).

Figure 9 visualizes inference results for the `Social1` trace. The figure consists of six bands (from up to down): measured power curve, uplink packets (each vertical line corresponds to one packet), downlink packets, bursts (§5.2), RRC states inferred by packets (with tails inferred too), and states inferred by power. Different RRC states are visualized by blocks with different shapes, shades, and colors. Each arrow on the timeline indicates a state transition

[5]As shown in Figure 8, the base power of TyTn II is 800 mW as long as the handset is on.

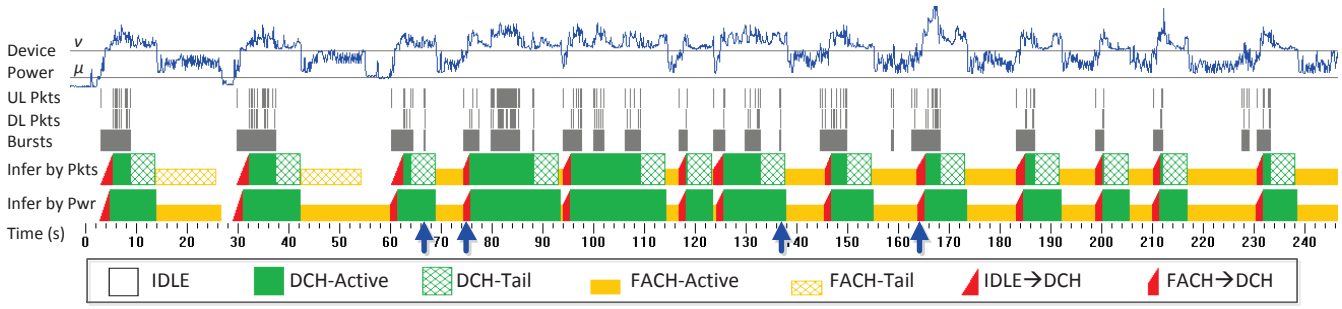

Figure 9: Comparing power-based and packet-based state inference results (the `Social1` trace). Each arrow on the timeline indicates a state transition that is misidentified by previous approach but is correctly identified by our inference algorithm.

Table 3: Measured average radio power consumption

	TyTn Carrier 1	NexusOne Carrier 1	ADP1 * T-Mobile
P(IDLE)	0	0	10mW
P(FACH)	460mW	450mW	401mW
P(DCH)	800mW	600mW	570mW
P(FACH→DCH)	700mW	550mW	N/A
P(IDLE→DCH)	550mW	530mW	N/A

* Reported by [31] on an Android HTC Dream phone

Table 4: TCP analysis: transport-layer properties of packets

Category	Label	Description
TCP connection management	`ESTABLISH`	A packet containing the SYN flag
	`CLOSE`	A packet containing the FIN flag
	`RESET`	A packet containing the RST flag
Normal data transfer	`DATA`	A normal data packet with payload
	`ACK`	A normal ACK packet without payload
TCP congestion, loss, and recovery	`DATA_DUP`	A duplicate data packet
	`DATA_RECOVER`	A data pkt echoing a duplicate ACK
	`ACK_DUP`	A duplicate ACK packet
	`ACK_RECOVER`	An ACK echoing a duplicate data pkt
Others	`TCP_OTHER`	Other special TCP packets

that is misidentified by the previous approach [24] but is correctly identified by our inference algorithm.

4.4 Applying the Radio Power Model

One important feature of ARO is the accurate estimation of radio energy consumption by using the inferred RRC states. Table 3 reports measured average radio power for an HTC TyTn II and a Google Nexus One for each RRC state and during each state promotion. Assuming these average values are representative [31, 24] despite other factors such as signal strength that also directly impact power consumption [26], the state inference results enable ARO to accurately profile radio energy consumption. Given the state inference results $S(t)$ and the measured radio power values $P(\cdot)$ shown in Table 3, the radio energy consumed between t_1 and t_2 is computed as $\int_{t_1}^{t_2} P(S(t))dt$.

5. PROFILING MOBILE APPLICATIONS

This section details analyses at higher layers, in particular the transport layer and the application layer, using TCP and HTTP as examples due to their popularity. We further describe how ARO uses cross-layer analysis results to profile resource efficiency of smartphone applications.

5.1 TCP and HTTP Analysis

TCP and HTTP analysis serve as prerequisites for understanding traffic patterns created by the transport layer and the application layer. Our main focus is on TCP and HTTP, as the vast majority of smartphone applications use HTTP over TCP to transfer application-layer data [21]. A recent large-scale measurement study [16] using datasets from two separate campus wireless networks (3 days of traffic for 32,278 unique devices) indicates that 97% of handheld traffic is HTTP.

We first describe the TCP analysis. ARO extracts TCP flows, defined by tuples of {srcIP, srcPort, dstIP, dstPort} from the raw packet trace, and then infers the transport-layer property for each packet in each TCP flow. In particular, each TCP packet is assigned to one of the labels listed in Table 4. The labels can be classified into four categories covering the TCP protocol behavior: *(i)* connection management, *(ii)* normal data transfer, *(iii)* TCP congestion, loss, and recovery, and *(iv)* other special packets (*e.g.,* TCP keep alive and zero-window notification).

In the third category, `DATA_DUP` is usually caused by a retransmission timeout or fast retransmission, and `ACK_DUP` is triggered by an out-of-order or duplicate data packet. Duplicate packets indicate packet loss, congestion, or packet reordering that may degrade TCP performance. A `DATA_RECOVER` packet has its sequence number matching the ack number of previous duplicate ACK packets in the reverse direction, indicating the attempt of a handset to transmit a possibly lost uplink packet or a downlink lost packet finally arriving from the server. Similarly, the ack number of an `ACK_RECOVER` packet equals to the sequence number of some duplicate data packets plus one, indicating the recipient of a possibly lost data packet.

ARO subsequently performs HTTP analysis by reassembling TCP flows then following the HTTP protocol to parse the TCP flow data. HTTP analysis provides ARO with the precise knowledge of mappings between packets and HTTP requests or responses.

5.2 Burst Analysis

As described earlier, low efficiencies of radio resource and energy utilization are attributed to short traffic bursts carrying small amount of data. ARO employs novel algorithms to identify them and to infer which factor triggers each such burst by synthesizing analysis results of the RRC, TCP, HTTP, and user input layer. Such triggering factors, which to our knowledge are not explored by previous effort, are crucial for understanding the root cause of inefficient resource utilization.

ARO defines a *burst* as consecutive packets whose inter-arrival time is less than a threshold δ. We set δ to 1.5 seconds since it is longer than commonly observed cellular round trip times [19]. Since state promotion delays are usually greater than δ, all state promotions detected in §4.2 are removed before bursts are identified. Each bar in the "Bursts" band in Figure 9 is a burst.

A burst can be triggered by various factors. Understanding them

Table 5: Burst Analysis: triggering factors of bursts

Label	The burst is triggered by ...
USER_INPUT	User interaction
LARGE_BURST	(The large burst is resource efficient)
TCP_CONTROL	TCP control packets (*e.g.,* FIN and RST)
SVR_NET_DELAY	Server or network delay
TCP_LOSS_RECOVER	TCP congestion / loss control
NON_TARGET	Other applications not to be profiled
APP	The application itself
APP_PERIOD	Periodic data transfers (One special type of APP)

benefits application developers who can then customize optimization strategies for each factor, *e.g.,* to eliminate a burst, to batch multiple bursts, or to make certain bursts appear less frequently. Some bursts are found to be inherent to the application behavior. We next describe ARO's burst analysis algorithm that assigns to each burst a triggering factor shown in Table 5 by correlating TCP analysis results and user input events.

The algorithm listed in Figure 10 consists of seven tests each identifying a triggering factor by examining burst size (duration), user input events, payload size, packet direction, and TCP properties (§5.1) associated with a burst. We explain each test as follows. A burst can be generated by a non-target application not profiled by ARO (Test 1). For Test 2, if a burst is large and long enough (determined by two thresholds th_s and th_d), it is assigned a LARGE_BURST label so ARO considers it as a resource-efficient burst. If a burst only contains TCP control packets without user payload (Lines 06 to 08), then it is a TCP_CONTROL burst as determined by Test 3. To reveal delays caused by server, network, congestion or loss, the algorithm then considers properties of the first packet in the burst in Test 4 and 5. For Test 6, if any user input activity is captured within a time window of ω seconds before a burst starts, then the burst is assigned a USER_INPUT label, if it contains user payload. For bursts whose triggering factors are not identified by the above tests, they are considered to be issued by the application itself (APP in Test 7). Most such bursts turn out (and are validated) to be periodic transfers (APP_PERIOD) triggered by an application using a software timer. We devise a separate algorithm to detect them (§5.2.1). In practice it is rare that a short burst satisfies multiple tests.

The burst analysis algorithm involves three parameters: th_s and th_d that quantitatively determine a large burst (Test 2), and the time window ω (Test 6). We set th_s = 100 KB, th_d = 5 sec, and ω = 1 sec. We empirically found that varying their values by $\pm 25\%$ (and $\pm 50\%$ for ω) does not qualitatively affect the analysis results presented in §7.

Within aforementioned seven tests, Test 1 to 3 are trivial. We validate Test 4 and 5 by setting up a web server and intentionally injecting server delay and packet losses. Evaluation for Test 6 and Test 7, which is more challenging due to a lack of ground truth, is done by manually inspecting our collected traces used for case studies (§7).

5.2.1 Identifying Periodic Transfers

We design a separate algorithm to spot APP_PERIOD bursts (Table 5), which are data transfers periodically issued by a handset application using a software timer. Such transfers are important because their impact on resource utilization can be significant although they may carry very little actual user data (*e.g.,* the Pandora application described in §7.2.1).

ARO focuses on detecting three types of commonly observed periodic transfers, though not mutually exclusive. They constitute the most simple forms of periodic transfers a mobile application can

```
01 Burst_Analysis (Burst b) {
02   Remove packets of non-target apps;
03   if (no packet left) {return NON_TARGET;} Test 1
04   if (b.payload > th_s && b.duration > th_d) Test 2
05     {return LARGE_BURST;}
06   if (b.payload == 0) { Test 3
07     if (b contains any of ESTABLISH, CLOSE, RESET,
08     TCP_OTHER packets)
09       {return TCP_CONTROL;}
10   }
11   d_0 ← direction of the first packet of b;
12   i_0 ← TCP label of the first packet of b;
13   if (d_0 == DL && (i_0 == DATA || i_0 == ACK)) Test 4
14     {return SVR_NET_DELAY;}
15   if (i_0 == ACK_DUP && i_0 == ACK_RECOVER &&
16   i_0 == DATA_DUP && i_0 == DATA_RECOVER) Test 5
17     {return TCP_LOSS_RECOVER;}
18   if (b.payload > 0 && find user input before b) Test 6
19     {return USER_INPUT;}
20   if (b.payload > 0) {return APP;} Test 7
21   else {return UNKNOWN;}
22 }
```

Figure 10: The burst analysis algorithm

```
01 Detect_Periodic_Transfers (t_1, t_2, ..., t_n) {
02   C ← {(d, t_i, t_j) | d = t_j − t_i  ∀j > i};
03   Find the longest sequence
04     D = (d_1, x_1, y_1), ..., (d_m, x_m, y_m) in C s.t.
05     (1) y_1 = x_2, y_2 = x_3, ..., y_{m−1} = x_m, and
06     (2) max(d_i) − min(d_i) < p;
07   if m ≥ q return mean(d_1, ..., d_m);
08   else return "no periodic transfer found";
09 }
```

Figure 11: Algorithm for detecting periodic transfers

do using HTTP: *(i)* periodically fetching the same HTTP object, *(ii)* periodically connecting to the same IP address, and *(iii)* periodically fetching an HTTP object from the same host. Detecting other periodic activities can be trivially added to the proposed detection framework shown in Figure 11. Also we found that existing approaches for periodicity or clock detection (*e.g.,* DFT-based [23] and autocorrelation-based [29]) do not work well in our scenario where the number of samples is much fewer.

The algorithm, shown in Figure 11, takes as input a time series $t_1, ..., t_n$, and outputs the detected periodicity (*i.e.,* the cycle duration) if it exists. It enumerates all $n(n-1)/2$ possible intervals between t_i and t_j where $1 \leq i < j \leq n$ (Line 2), from which the longest sequence of intervals is computed by dynamic programming (Lines 3-6). Such intervals should be consecutive (Line 5) and have similar values whose differences are bounded by parameter p (Line 6). If the sequence length is long enough, larger than the threshold parameter q, then the average interval is reported as the cycle duration (Line 7). We empirically set p=1 sec and q=3 based on evaluating the algorithm on *(i)* randomly generated test data (periodic time series mixed with noise), and *(ii)* real traces studied in §7.

5.3 Profiling Applications

We describe how ARO profiles mobile applications using cross-layer analysis. First, leveraging RRC state inference and burst analysis results, ARO computes for each burst (with its triggering factor known) its radio resource and radio energy consumption. Then the TCP and HTTP analysis described in §5.1 allow ARO to associate each burst with the transport-layer or the application-layer behav-

ior so that an ARO user can learn quantitatively what causes the resource bottleneck for the application of interest.

We describe two methodologies for quantifying the resource consumption of one or more bursts of interest: computing the *upperbound* and the *lowerbound*. Their key difference is whether or not they consider non-interested bursts whose tails help reduce the resource consumption of those interested bursts.

Method 1: Compute the upperbound of resource consumption. The radio energy consumed by burst B_i is computed as $\int_{t_1}^{t_2} P(S(t))dt$ where $S(t)$ is the inferred RRC state at time t and $P(\cdot)$ is the power function (Table 3). t_1 is the time when Burst B_i starts consuming radio resources. Usually t_1 equals to the timestamp of the first packet of B_i. However, if B_i begins with a downlink packet triggering a state promotion, t_1 should be shifted backward by the promotion delay since radio resources are allocated during the state promotion before the first packet arrives (§4.2). t_2 is the timestamp of the first packet of the next burst B_{i+1}, as tail times incurred by B_i need to be considered (there may exist IDLE periods before t_2, but they do not consume any resource). Similarly, t_2 is shifted backward by the promotion delay if necessary. The radio resources consumed by B_i are quantified as the DCH occupation time between t_1 and t_2. We ignore radio resources allocated for shared low-speed FACH channels.

Method 2: Compute the lowerbound. One problem with Method 1 is that it may *overestimate* a burst's resource consumption, which may already be covered by the tail of a previous burst. For example, consider burst Y in Figure 12(a). Its resource utilization lasts from t=12.5 sec to t=18.3 sec according to Method 1. However, such an interval is already covered by the tail of the previous burst. In other words, the overall resource consumption is not reduced even if in the absence of burst Y.

To address this issue, we propose another way to quantify the resource impact of one or more bursts by computing the *difference* between the resource consumption of two scenarios where the bursts of interest are kept and removed, respectively. For example, in Figure 12, let X and Y be the bursts of interest. Trace (a) and (d) correspond to the original trace and a modified trace where X and Y are removed. Then their energy impact is computed as $E_a - E_d$ where E_a and E_d correspond to the radio energy consumption of trace (a) and (d), respectively. The consumed resource computed by this method does not exceed that computed by Method 1.

5.3.1 Modifying Cellular Traces

The aforementioned Method 2 is intuitive, while the challenge here is to construct a trace with some packets removed. In particular, RRC state promotion delays affect the packet timing. Therefore, removing packets directly from the original trace causes inaccuracies as it is difficult to transform the original promotion delays to promotion delays in the modified trace with different state transitions. To address such a challenge, we propose a novel technique for modifying cellular traces. The high-level idea is to first *decouple* state promotion delays from application traffic patterns before modifying the trace, then reconstruct the RRC states for the modified trace.

The whole procedure is illustrated in Figure 12a-d (assuming we want to remove bursts X are Y). First, the original trace (Figure 12a) is *normalized* by removing all promotion delays (Figure 12b). This essentially decouples the impact of state promotions from the real application traffic patterns [24]. Then the bursts of interest are removed from the normalized trace (Figure 12c). Next, ARO runs the state inference algorithm again to reconstruct the RRC states with state promotions injected using the average promotion delay values shown in Table 1 (Figure 12d). As expected,

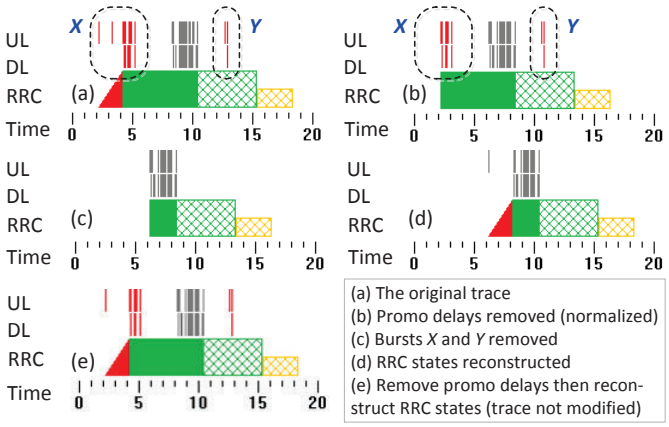

Figure 12: An example of modifying cellular traces (X and Y are the bursts of interest to be removed)

the first packet in Figure 12d triggers a promotion that does not exist in the original trace (a).

Validation. We demonstrate the validity of the proposed cellular trace modification technique as follows. For each of the 40 traces listed in Table 6, we compare the state inference results for *(i)* the original trace (*e.g.,* Figure 12a) and *(ii)* a trace with promotion delays removed then RRC states reconstructed, but without any packet removed (*e.g.,* Figure 12e). Ideally their RRC inference results should be the same. Our comparison results show that for each of the 40 traces, both inference results are almost identical as their time overlap (defined in §4.3) is at least 99%, and their total radio energy consumption values differ by no more than 1%. The small error stems from the difference between original promotion delays and injected new promotion delays using fixed average values. This demonstrates that the algorithm faithfully reconstructs the RRC states. In §7, we show resource consumption computed by both Method 1 and Method 2.

6. IMPLEMENTATION

We briefly describe how we implemented ARO. We built the data collector on Android 2.2 by adding two new features (1K LoC) to tcpdump: logging user inputs and finding packet-to-application correspondence (§3). ARO reads `/dev/input/event*` that captures all user input events such as touching the screen, pressing buttons, and manipulating the tracking ball.

Finding the packet-to-application correspondence is more challenging. The ARO data collector realizes this using information from three sources in Android OS: `/proc/PID/fd` containing mappings from process ID (PID) to inode of each TCP/UDP socket, `/proc/net/tcp(udp)` maintaining socket to inode mappings, and `/proc/PID/cmdline` that has the process name of each PID. Therefore socket to process name mappings, to be identified by the data collector, can be obtained by correlating the above three pieces of information. Doing so once for all sockets takes about 15 ms on Nexus One, but it is performed only when the data collector observes a packet belonging to a newly created socket or the last query times out (we use 30 seconds).

The runtime overhead of the data collector mainly comes from capturing and storing the packet trace. When the throughput is as high as 600 kbps, the CPU utilization of the data collector can reach 15% on Nexus One although the overhead is much lower when the throughput is low. There is no noticeable degradation of user experience when the data collector is running.

The analyzers were implemented in C++ on Windows 7 (7.5K LoC). The analysis time for the entire workflow shown in Figure 3 is usually less than 5 seconds for a 10-minute trace. As mentioned in §3, ARO configures the RRC analyzer with handset and carrier specific parameters. Currently our ARO prototype supports one carrier and two types of handsets (Table 1 and Table 3). The RRC analyzer for other carriers can be designed in a way very similar to §4.1. Differences among handsets mainly lie in radio power consumption and the fast dormancy behavior that are easy to measure. Also note that TCP, HTTP, and burst analyzers are independent of specific handset or carrier.

7. ARO USE CASE STUDIES

To demonstrate typical usage scenarios of ARO, we present case studies of six real Android applications and show their resource inefficiencies identified by ARO. All applications described in this section are in the "Top Free" section of Android Market and have been downloaded at least 250,000 times as of December 2010. The handset used for experiments is a Google Nexus One phone with fast dormancy (its α and β timers are 5 sec and 3 sec, respectively as shown in Figure 7). For identified inefficiencies, their resource waste is even higher if fast dormancy is not used. All experiments were performed between September 2010 and November 2010 using Carrier 1's UMTS network whose RRC state machine is depicted in Figure 1.

7.1 Experimental Methodology

Our experimental methodology is straightforward: for each application studied, we collected a trace by running the application for at least 5 minutes (except for Google Search), then used ARO to analyze the trace. Several factors including user behavior randomness, traffic of non-target applications, and radio link quality, may affect the data collected and henceforth the analysis results. Clearly, the discovered traffic patterns should be *stable* in that they are inherent to the application logic and thus are not affected by user behavior randomness in common application usage scenarios. To ensure this, for each application, we analyzed at least 5 traces collected by at least 3 students who used the application as normal users (except for Pandora and Mobclix that do not involve user interaction). A case listed in Table 6 was reported only if the same symptom was observed in *all* collected traces so that we had high confidence that the observed traffic patterns stemmed intrinsically from the application logic (although it is still useful to learn uncommon problems from individual traces).

To minimize the impact of non-target applications that concurrently access the network, we discarded a trace if any of the transferred bytes, the radio energy, or the DCH time caused by `NON_TARGET` bursts (Table 5) was greater than 5% of the total bytes, the total radio energy, or the total DCH time, respectively. We did similar filtering by examining `TCP_LOSS_RECOVER` bursts. Further, to minimize the impact of poor radio link quality, we collected all traces at reasonable signal strength conditions.

7.2 Results

We now describe case studies for the six applications. As shown in Table 6, for each application, we also found other popular applications with the same problem identified by ARO. The last column of Table 6 shows the layers that are related to the identified inefficiency. *RRC*, *TCP*, *App*, and *User* correspond to the RRC layer, the transport layer, the application layer, and the user input layer, respectively.

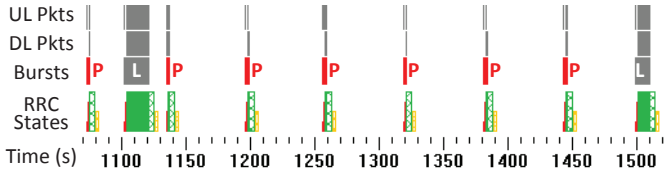

Figure 13: Pandora visualization results. "L" (grey) and "P" (red) bursts are `LARGE_BURST` and `APP_PERIOD` bursts, respectively.

Table 7: Pandora profiling results (Trace len: 1.45 hours)

Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
`LARGE_BURST`	96.4%	35.6%	35.9%	42.4%	42.5%
`APP_PERIOD`	0.2%	45.9%	46.7%	40.4%	40.9%
`APP`	3.2%	12.8%	13.4%	12.4%	12.8%
`TCP_CONTROL`	0.0%	1.2%	1.6%	1.1%	1.5%
`TCP_LOSS_RECOVER`	0.2%	0.2%	0.6%	0.3%	0.7%
`NON_TARGET`	0.0%	1.8%	1.8%	1.7%	1.7%
Total	23.6 MB	846 J		895 sec	

7.2.1 *Pandora Streaming*

Pandora is a popular music streaming application. We collected three Pandora traces by simply listening to the music for at least 1 hour for each trace. Table 7 shows the profiling results of one trace, and the results for other traces are qualitatively similar. The "UB" (upperbound) and "LB" (lowerbound) columns in Table 7 refer to the resource consumption computed by Method 1 and 2 described in §5.3, respectively. Numbers in the "LB" column do not necessarily add up to 100%.

High resource overhead of periodic audience measurements. The profiling results in Table 7 indicate that periodic data transfers (`APP_PERIOD` bursts), which carry only 0.2% of total bytes, account for 46% of total radio energy consumption and 40% of radio resource usage. The detection algorithm in Figure 11 further pinpoints that for every 62.5 seconds, Pandora connects to `lt.andomedia.com`, which provides various real-time audience measurement services (*e.g.,* monitor online listeners' favorite radio stations), and downloads hundreds of bytes. Each such a burst, however, triggers an IDLE→DCH promotion and subsequently two tails of 8 seconds in total. On the other hand, Pandora usually takes less than 30 seconds to download a song (a `LARGE_BURST` burst) that can be played for several minutes. As illustrated in Figure 13, the total DCH occupation time for periodic data transfers is similar to or even longer than the music streaming time, although the former carries much fewer and much less important user data than the latter. The `APP` bursts shown in Table 7 correspond to non-periodic transfers of album images and other metadata. Here one straightforward fix is to increase the periodicity. A more intelligent approach is to batch such delay-tolerant transfers with delay-sensitive transfers to reduce the overall tail time [10].

7.2.2 *Fox News*

Fox News is a popular news application. We obtained five traces from three users who browsed the news headlines or articles as they like for at least 5 minutes. Table 8 exemplifies profiling results of one representative trace. The results indicate that ARO is essential in quantitatively breaking down resource consumption into bursts with their triggering factors inferred.

Scattered bursts due to scrolling. Table 8 indicates that the majority of resources are spent on bursts initiated by user interactions. Among them, about 15%∼18% of radio energy is re-

Table 6: Case studies of six popular Android applications

App name	App Mode	# traces	Case Description (Recommendations)	Similar apps	Layers
Pandora §7.2.1	Background	3	High resource overhead of periodic audience measurements (Delay transfers and batch them with delay-sensitive transfers)	Fox News Tune-in Radio	RRC, App
Fox News §7.2.2	Foreground	5	Scattered bursts due to scrolling (Transfer them in one burst)	USA Today	RRC, App, User
			Transferring duplicated contents (Use the "Expires" HTTP header)	NY Times	App
BBC News §7.2.3	Foreground	10	Inefficient content prefetching (Use HTTP pipelining for transferring multiple small objects for networks with high bw-delay product)	NY Times	TCP, App
			Scattered bursts of delayed FIN/RST packets (Close a connection immediately if possible, or within tail time)	CBS News Google Shopper	RRC, TCP, App
Google Search §7.2.4	Foreground	15	High resource overhead of query suggestions and instant search (Balance between functionality and resource when battery is low)	Bing Search Yahoo Search	RRC, App, User
Tune-in Radio §7.2.5	Foreground	5	Low DCH utilization due to constant-bitrate streaming (Buffer data and periodically stream data in one burst)	NPR Radio Iheartradio Radio	RRC, App
Mobclix §7.2.6	Foreground	2	Aggressive ad refresh rate making a handset persistently occupy FACH or DCH (decrease the refresh rate, piggyback or batch ad updates)	Apps with Mobclix ads embedded	RRC, App

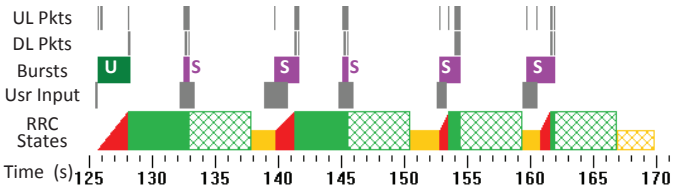

Figure 14: The Fox News results. "U" (green) and "S" (purple) bursts are triggered by tapping and scrolling the screen, respectively.

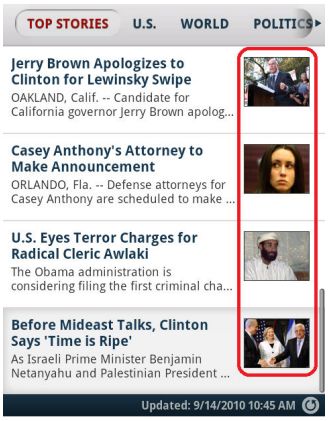

Figure 15: Headlines of the Fox News application. The thumbnail images (highlighted by the red box) are transferred only when they are displayed as a user scrolls down the screen.

sponsible for bursts generated when a user scrolls the screen. By examining HTTP responses associated with such bursts, we discover that thumbnail images embedded in headlines (Figure 15) are transferred only when they are displayed as a user scrolls down the screen. Thus as illustrated in Figure 14, when a user browses the headlines, the handset always occupies the DCH state due to such on-demand transfers of thumbnails. On the other hand, each thumbnail has very small size (less than 5KB each). A suggested improvement is to download all thumbnails (usually less than 15) in one burst. Doing so significantly shortens the overall DCH occupation time for headline browsing with negligible bandwidth overhead incurred. We observe this problem for other news applications (*e.g.,* USA Today) that use the same application framework.

Transferring duplicate contents. The HTTP analyzer (§5.1) extracts HTTP objects from the trace. We discovered that often,

Table 8: Fox News profiling results (Trace len: 10 mins)

Prefetching Phase					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
USER_INPUT(Click)	91.0%	56.7%	67.6%	60.2%	70.4%
USER_INPUT(Scroll)	5.9%	15.2%	17.9%	14.7%	16.7%
APP_PERIOD	1.5%	5.2%	7.5%	6.1%	7.4%
TCP_CONTROL	0	0.7%	3.7%	0.0%	2.3%
TCP_LOSS_RECOVER	1.5%	0.7%	2.5%	1.9%	3.2%
SVR_NET_DELAY	0.1%	0.4%	0.8%	0.0%	0.0%
Total	1.0 MB	276 J		284 sec	

Table 9: BBC News profiling results

Prefetching Phase (1.4 mins)					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
LARGE_BURST	100%	100%	100%	100%	100%
Total	1.1 MB	60.1 J		82.8 sec	
User-triggered Fetching Phase (8 mins)					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
TCP_CONTROL	0	11.3%	24.2%	0.0%	5.7%
USER_INPUT	98.7%	42.5%	73.1%	37.9%	90.0%
SVR_NET_DELAY	1%	0.0%	2.7%	0.0%	5.2%
Total	162 KB	145 J		120 sec	

the same content is repeatedly transferred, leading to waste of bandwidth. For example, Fox News fetches the same object `foxnews.com/weather/feed/getWeatherXml` whenever a news article is loaded, and the response from the server (45 KB) is identical unless the weather information, updated hourly, changes. The problem can be fixed by letting the server put an "Expires" header in an HTTP response to explicitly tell the client how long the content can be cached [3].

7.2.3 BBC News

BBC News is another news application. Unlike Fox News, which fetches an article only when a user wants to read it, the network usage of BBC News consists of two phases: prefetching and user-triggered data fetching.

Inefficient content prefetching. Prefetching happens when a news category (*e.g.,* Sports), which is not yet cached or is out-of-date, is selected by a user. In the prefetching phase, the application downloads the headline page with thumbnails, and more aggressively, contents of all articles of the selected news category in a single large burst. While it is arguable whether aggressive prefetch-

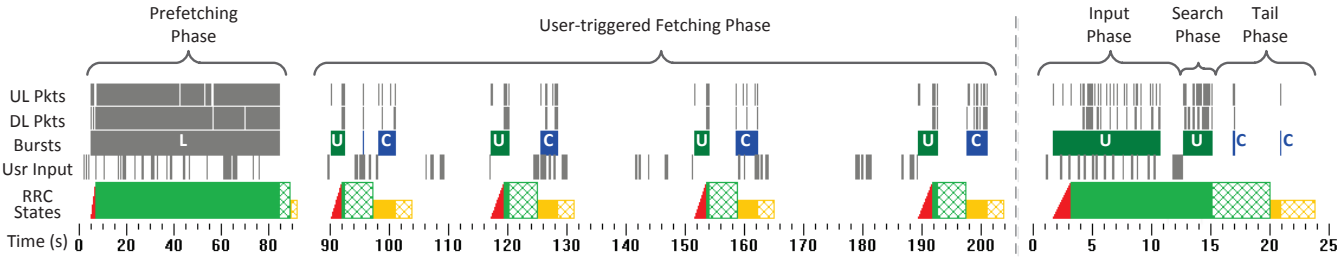

Figure 16: BBC News results: prefetching followed by 4 user-triggered transfers. "U" (green), "C" (blue), and "L" (grey) bursts are `USER_INPUT`, `TCP_CONTROL`, and `LARGE_BURST` bursts, respectively.

Figure 17: ARO visualization results for Google search

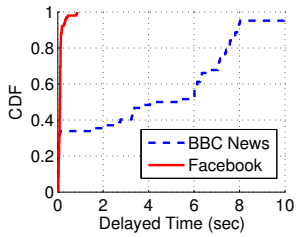

Figure 18: Distribution of delayed time for FIN or RST packets for BBC News and Facebook applications

ing, which efficiently utilizes radio resources but wastes network bandwidth as some contents may not be consumed by end users, is a good strategy, the prefetching of BBC News is performed inefficiently. It takes up to two minutes for BBC News to prefetch all articles (*e.g.,* 60 articles in one trace) of a news category. The HTTP analyzer reveals that the application issues one single HTTP GET for each article, then waits for the response before issuing the next HTTP GET. A more efficient approach is HTTP pipelining, *i.e.,* the application sends all 60 URLs in a single HTTP GET and hence the server transfers all articles without interruption. Given the scenario where many small objects are transferred in a network of high bandwidth-delay product, HTTP pipelining, which is widely supported by modern web servers, dramatically improves the throughput by eliminating unnecessary round trips and allowing more outstanding (*i.e.,* in-flight) data packets with almost no head-of-line blocking overhead [11].

Scattered bursts due to delayed FIN/RST. After prefetching, clicking on an article triggers very little traffic. However, as shown in Table 9, `TCP_CONTROL` bursts, which do not carry any user payload, consume 11%∼24% of the radio energy. Such `TCP_CONTROL` bursts are FIN or RST packets, *i.e.,* the application delays closing TCP connections. As shown in Figure 16, they waste radio energy by causing additional FACH occupation time.

Delayed FIN or RST packets are caused by connection timeout maintained by either an HTTP client or server that uses persistent HTTP connections. Different applications may use different timeout values since the HTTP 1.1 protocol places no requirements on how to set the value [15]. We observe that some applications (*e.g.,* Facebook and Amazon Shopper) always immediately shut down a connection, while BBC News may delay closing a connection by up to 15 seconds after the last HTTP response is transmitted. In our traces, 50% of its FIN/RST are delayed by at least 5 seconds, which is the α timer value, potentially triggering a FACH→DCH promotion.

Figure 18 plots distributions of delayed time for FIN or RST packets for two application traces. Facebook always immediately shuts down a connection, while BBC News may delay closing a connection by up to 15 seconds after the last HTTP response is transmitted. We observe from traces that most FIN and RST packets are initiated by a handset instead of by a server.

Eliminating delayed FIN/RST packets saves resources, but closing a connection too early may prevent it from being reused, thus incurring additional overhead for establishing new connections. A compromise is to close the connection before the α timer expires to avoid a state promotion triggered by delayed FIN/RST. For Carrier 1, doing so further benefits handset battery life, as usually FIN and RST packets do not reset the α timer due to their small sizes (§4.1). Smartphone OS can help applications properly close TCP connections by collecting hints from applications and employing different connection timeout values depending on the carrier type.

7.2.4 *Google Search*

Search is among the most popular browsing activities on smartphones [14]. Almost all search engines provide real-time query suggestions as a user types keywords in the search box. We show that such a feature consumes significant radio energy (up to 78%) and radio resources (up to 76%) by conducting a user study.

Five student users participated in our user study. Each student searched three keywords in mobile version of Google using Nexus One: "university of michigan", "ann arbor", and "android 2.2". A trial is abandoned if any typing mistake was made (typing mistakes worsen the resource efficiency). The participants were asked to use the query suggestion whenever possible. Browser caches were cleared before each trial. We believe these keywords are representative although the length and popularity of keywords may affect the results.

High resource overhead of real-time query suggestions and instant search. We obtained 15 traces (3 keywords searched by 5 users) which were further analyzed by ARO. We broke down each trace into three phases: *(i)* Input Phase, *i.e.,* a user is typing a keyword. *(ii)* Search Phase, *i.e.,* after a user submits the keyword, and before the last byte of the search results is received. *(iii)* Tail Phase, *i.e.,* the remaining time until the RRC state is demoted to IDLE. An example for searching "university of michigan" is shown in Figure 17. Subsequently, ARO computes transferred payload bytes (Figure 19-a), radio energy consumption (Figure 19-b), and DCH time (Figure 19-c) for each phase. Each plot of Figure 19 consists of results of the three keywords. For each keyword, "I", "S", and "T" correspond to Input Phase, Search Phase, and Tail Phase, respectively. Figures 17 and 19 clearly show that while a user is typing a keyword, real-time query suggestions keep the handset at DCH, consuming 2.3 to 3.5 times of radio energy and 1.8 to 3.2 times of DCH time, compared to those consumed by Search Phase. We note that a similar problem occurs for Google instant search (results appear instantly as a user types a keyword) that is available for Android since Nov 2010 [4].

Query suggestions and instant search improve user experience. However, realizing their high resource impact in cellular network,

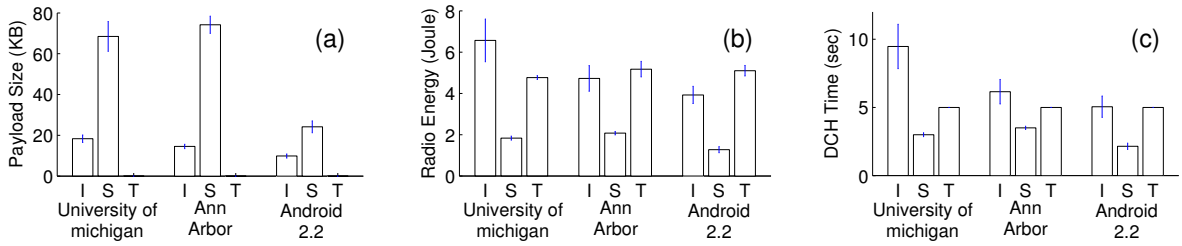

Figure 19: Breakdown of (a) transferred payload size (b) radio energy (c) DCH occupation time for searching three keywords in Google. "I", "S", "T" correspond to Input Phase, Searching Phase, and Tail Phase, respectively.

Table 10: Constant bitrate vs. bursty streaming

Name	Server	bitrate	Radio Power
NPR News	SHOUTcast	32 kbps$^+$	36 J/min
Tune-in	Icecast	119 kbps	36 J/min
Iheartradio	QTSS	32 kbps	36 J/min
Pandora	Apache	bursty	11.2 J/min
Pandora w/o mes*	Apache	bursty	4.8 J/min
Slacker	Apache	bursty	10.9 J/min

*A hypothetical case where all periodic audience measurement data transfers are removed.
$^+$NPR News also uses a higher bitrate of 128 kbps for some content.

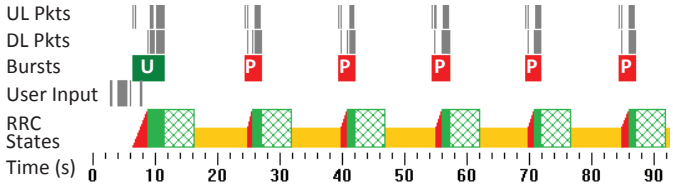

Figure 20: Results for Mobclix (w/o FD). "U" (green) and "P" (red) bursts are `USER_INPUT` and `APP_PERIOD` bursts, respectively.

the application can balance between functionality and resource when the latter becomes a bottleneck (*e.g.,* the battery is critically low). For example, using historical keywords and a local dictionary to suggest search hints is an alternative but with worse functionality.

7.2.5 *Tune-in Radio (and Other Streaming Apps)*

The Tune-in Radio application delivers live streams of hundreds of FM/AM radio stations. Table 10 further lists NPR News and Iheartradio, two popular live radio streaming applications similar to Tune-in Radio. All three applications employ existing radio streaming schemes that work well on wired networks and WiFi: the server streams data at a constant bitrate (*e.g.,* 32 kbps) to a client without any pause.

Low DCH utilization due to constant-bitrate streaming. In cellular networks, however, continuously streaming at a constant low bitrate causes considerable inefficiencies on resource utilization, as a handset is always using the DCH channel, whose available bandwidth is significantly under-utilized, whenever a user is listening to the radio. Table 10 compares constant-bitrate streaming to the bursty streaming strategy employed by Pandora and Slacker Radio where a program is buffered in one burst utilizing the maximum available bandwidth then the application does not access the network while playing the program. The last column of Table 10 indicates that for the two streaming strategies, their energy efficiency, *i.e.,* the average radio energy consumption for listening to the radio for 1 minute, differs by up to 7.5 times. For radio programs whose real-time is not strictly required (*e.g.,* their delivery can be delayed by one minute), a live streaming server can also perform similar bursty streaming to save handset energy and radio resources by buffering data and periodically streaming data in one burst.

7.2.6 *Mobclix (and Other Mobile Ad Platforms)*

We investigate advertisement dissemination strategies for three popular mobile ad platforms: Google Mobile Ad, AdMob, and Mobclix. All platforms allow developers to easily display ads in their applications by providing simple SDKs. We thus built three toy Android applications each using one ad platform with its default configuration. Then we employed ARO to profile each ad-embedded application for five hours, using two Nexus One phones (Table 1) where one has a shorter FACH→IDLE timer (3 sec) due to fast dormancy (the "w/ FD" column in Table 11) and the other uses default timers of Carrier 1 without fast dormancy (the "w/o FD" column). Also note that our toy applications themselves do not have any network activity so the network traffic is solely generated by ad modules.

Table 11: Comparing three mobile ad platforms

Name	Default Refresh Rate	Avg Update Size	Radio Power	
			w/ FD	w/o FD
Google Mobile Ad	180.0 sec	6.0 KB	2.5 J/min	3.6 J/min
AdMob	62.5 sec	6.8 KB	5.7 J/min	8.8 J/min
Mobclix	15.0 sec	1.4 KB	23.2 J/min	29.6 J/min

Aggressive ad refresh rate. We discuss the profiling results. As identified by ARO, all ad platforms by default employ a fixed refresh rate for updating the ads. For example, an application using AdMob pings `r.admob.com` for every 62.5 sec. Then the server may either push a new ad or let the application display the existing ad. Surprisingly, as summarized in Table 11, the three platforms use considerably different refresh rates, leading to remarkable disparity of their radio power consumption, especially for applications without network activities (*e.g.,* games). In particular, as illustrated in Figure 20, Mobclix employs an aggressive refresh rate of 15 sec that is even shorter than the default tail time of Carrier 1 (17 sec when fast dormancy is not used), making the handset persistently occupying DCH or FACH whenever the application is running.

8. RELATED WORK

We describe related work in three categories below.

Profiling and measurements. Previous work [24] systematically characterizes the impact of the RRC state machine on radio resources and energy by analyzing traces collected from a commercial UMTS network. Similar measurements have been done by [30, 20] using analytical models. Recent work [13] also investigates impact of traffic patterns on radio power management policy and proposes suggestions such as reducing the tail time to save handset energy. These studies did examine the interplay between smartphone

applications and the state machine behavior, while our work makes a significant further step by introducing a novel tool that systematically correlates information at multiple layers to reveal the low efficiency of resource utilization to application developers. Other measurement work (*e.g.,* 3gTest [19], LiveLab [28], and [14]) collect traces from real smartphone users, focusing on characterization at only IP and higher layers. Complementary to our work are profiling tools (*e.g.,* PowerTutor [31]) that focus on power modeling.

RRC Inference techniques. Previous work [22] introduced 3G Transition Triggering Tool to infer RRC state machine parameters. [24] further considered and inferred different state transition models configured by two commercial UMTS carriers. Previous effort [24, 25] also uses a simulation-based approach to obtain RRC state machine statistics. Our inference algorithm presented in this paper differs from the previous approach with different scope of application, finer simulation granularity, and higher accuracy (§4).

Optimizing traffic patterns. Many work propose techniques to optimize smartphone application performance and energy efficiency. [10] introduced TailEnder, which delays transfers of delay-tolerant traffic and batches them with normal traffic, so that the overall tail time incurred by delay-tolerant traffic could be reduced. Also prefetching could be used to reduce tail time. Similar batching strategies are presented in [9] and [26]. In contrast, ARO provides application developers with more opportunities to optimize short bursts that can be triggered by multiple factors.

9. CONCLUDING REMARKS

We have presented ARO, the first tool that exposes the cross-layer interaction for layers ranging from radio resource control to application layer. We have demonstrated using popular mobile applications that ARO helps reveal several previously unknown, general categories of inefficient resource usage, affecting distinct classes of mobile applications due to a lack of transparency in the lower-layer protocol behaviors. In particular, we are starting to contact developers of popular applications such as Pandora. The feedback has been encouragingly positive as the provided technique greatly helps developers identify resource usage inefficiencies and improve their applications [2]. We are actively working on releasing our tool to smartphone users and developers.

Our work opens and enables new research opportunities for *(i)* analyzing other cross-layer information (*e.g.,* cellular handoff events, OS events, and application activities) for more fine-grained diagnosis of performance and resource utilization inefficiencies, and *(ii)* providing automated mitigation solutions to identified inefficiencies. We plan to explore them in our future work.

10. ACKNOWLEDGEMENTS

We appreciate the valuable suggestions from Yudong Gao, Zhiyun Qian, and Doug Sillars for improving our tool. We would also like to thank Mobisys reviewers and especially Aruna Balasubramanian for shepherding the paper. This research was sponsored in part by NSF grant #CNS-0643612, and by Navy award N00014-09-1-0705, and by Army award W911NF-08-1-0367.

11. REFERENCES

[1] 300 Million UMTS Subscribers. http://www.3gpp.org/300-million-UMTS-subscribers.

[2] A Call for More Energy-Efficient Apps. http://www.research.att.com/articles/featured_stories/2011_03/201102_Energy_efficient.

[3] Add an Expires or a Cache-Control Header. http://developer.yahoo.com/performance/rules.html#expires.

[4] Google Instant search now available for iOS4 and Android 2.2+. http://www.mobileburn.com/news.jsp?Id=12012.

[5] Monsoon Power Monitor. http://www.msoon.com/.

[6] GERAN RRC State Mchine. 3GPP GAHW-000027, 2000.

[7] Configuration of Fast Dormancy in Release 8. 3GPP discussion and decision notes RP-090960, 2009.

[8] System Impact of Poor Proprietary Fast Dormancy. 3GPP discussion and decision notes RP-090941, 2009.

[9] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *Mobisys*, 2010.

[10] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[11] R. Chakravorty and I. Pratt. WWW Performance over GPRS. In *IEEE MWCN*, 2002.

[12] M. Chatterjee and S. K. Das. Optimal MAC State Switching for CDMA2000 Networks. In *INFOCOM*, 2002.

[13] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *IMC*, 2010.

[14] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, and R. G. D. Estrin. Diversity in Smartphone Usage. In *Mobisys*, 2010.

[15] R. Fielding, J. Gettys, J. Mogul, H. F. L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1 . RFC 2616, 1999.

[16] A. Gember, A. Anand, and A. Akella. A Comparative Study of Handheld and Non-Handheld Traffic in Campus WiFi Networks. In *PAM*, 2011.

[17] H. Holma and A. Toskala. HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications. John Wiley and Sons, Inc., 2006.

[18] H. Holma and A. Toskala. WCDMA for UMTS: HSPA Evolution and LTE. John Wiley and Sons, Inc., 2007.

[19] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *Mobisys*, 2010.

[20] F. Liers, C. Burkhardt, and A. Mitschele-Thiel. Static RRC Timeouts for Various Traffic Scenarios. In *PIMRC*, 2007.

[21] G. Maier, F. Schneider, , and A. Feldmann. A First Look at Mobile Hand-held Device Traffic. In *PAM*, 2010.

[22] P. Perala, A. Barbuzzi, G. Boggia, and K. Pentikousis. Theory and Practice of RRC State Transitions in UMTS Networks. In *Proc. of IEEE Broadband Wireless Access Workshop*, 2009.

[23] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP Revisited: A Fresh Look at TCP in the Wild. In *IMC*, 2009.

[24] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *IMC*, 2010.

[25] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation. In *ICNP*, 2010.

[26] 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 *Mobicom*, 2010.

[27] S. Sesia, I. Toufik, and M. Baker. LTE: The UMTS Long Term Evolution From Theory to Practice. John Wiley and Sons, Inc., 2009.

[28] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: Measuring Wireless Networks and Smartphone Users in the Field. In *HotMetrics*, 2010.

[29] B. Veal, K. Li, and D. Lowenthal. New Methods for Passive Estimation of TCP Round-Trip Times. In *PAM*, 2005.

[30] J.-H. Yeh, J.-C. Chen, and C.-C. Lee. Comparative Analysis of Energy-Saving Techniques in 3GPP and 3GPP2 Systems. *IEEE transactions on vehicular technology*, 58(1), 2009.

[31] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *CODES+ISSS*, 2010.