RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage

Pavan K. Athivarapu[1], Ranjita Bhagwan[1], Saikat Guha[1],
Vishnu Navda[1], Ramachandran Ramjee[1],
Dushyant Arora[2]*, Venkata N. Padmanabhan[1], George Varghese[3]*
[1]Microsoft Research India [2]Princeton University [3]UC San Diego

ABSTRACT

Many networked applications that run in the background on a mobile device incur significant energy drains when using the cellular radio interface for communication. This is mainly due to the radio-tail, where the cellular radio remaining in a high energy state for up to 20s after each communication spurt. In order to cut down energy consumption, many recent devices employ fast dormancy, a feature that forces the client radio to quickly go into a low energy state after a fixed short idle period. However, aggressive idle timer values for fast dormancy can increase signaling overhead due to frequent state transitions, which negatively impacts the network. In this work, we have designed and implemented RadioJockey, a system that uses program execution traces to predict the end of communication spurts, thereby accurately invoking fast dormancy without increasing network signaling load. We evaluate RadioJockey on a broad range of background applications and show that it achieves 20-40% energy savings with negligible increase in signaling overhead compared to fixed idle timer-based approaches.

Categories and Subject Descriptors

C.2.1 [**Computer Communications Networks**]: Network Architecture and Design—*Wireless communication*; C.4 [**Performance of Systems**]: Design Studies, Performance Attributes

General Terms

Algorithms, Design, Measurement, Experimentation, Performance

Keywords

Cellular, 3G, LTE, 4G, Energy Saving, Signaling Overhead, Fast Dormancy

1. INTRODUCTION

Several applications running on smartphones, tablets and laptops perform network activity while running in the background for tasks such as syncing state (e.g., emails, presence, newsfeeds, etc.). Often these tasks involve only a few seconds of communication interspersed between longer periods of inactivity. Nevertheless, several platforms (e.g., iPhone and Windows Phone) prohibit background applications or severely restrict them, primarily because of the significant energy costs of supporting such workloads over a cellular radio interface.

Even short network bursts have significant energy costs due to the intrinsic nature of cellular radio operation. At the beginning of each communication spurt, the cellular radio wakes up from a low power idle state and transitions to a high energy active state (DCH) to transfer packets. This DCH setup operation takes up to 30 signaling messages and about 2 seconds [9]. After the communication spurt ends, the radio continues to remain in DCH state for a certain inactive duration (e.g., 10-20s) before transitioning back to idle state. This inactive duration is determined by the network operator in both the 3G and the 4G LTE standards [15] and is commonly referred to as the *radio-tail* [2, 18]. A long radio-tail cuts down on repeated DCH setups and the attendant (unmonetized) signaling bandwidth costs. However, while long radio-tails help keep signaling costs low, they incur an energy penalty of up to 16J per tail and severely impact the energy consumption of background applications.

Many recent smartphone models cut down the energy cost of radio-tails by implementing a feature called *Fast Dormancy* (FD), which forces the radio to quickly transition from DCH to a low energy state (Idle or PCH). Fast dormancy is typically invoked with a fixed short inactivity timer (e.g., 3-5s), thereby reducing the energy consumed during the radio tail. However, as we show in Section 2, aggressive fixed idle timer values face two fundamental issues: i) some complex applications such as Outlook (email) have an inherently large variance in their inter-packet arrival distribution; and ii) even for simple applications, network conditions may change as the device moves, significantly altering the inter-packet arrival distribution. Since there is no fixed sweet-spot in inter-packet arrivals, the use of aggressive fixed idle timers can result in unpredictable increases in signaling load as the device repeatedly transitions between idle and DCH states. Indeed, in the early days of fast dormancy, popular devices employing aggressive timers lead to episodes of severe signaling channel congestion [1].

Fundamentally, there is an inherent trade-off between energy saved using fast dormancy and the signaling overhead generated by the device. To date, this trade-off has mostly been explored in an ad-hoc manner through the use of idle timers. Instead, in this paper, we have designed a system called *RadioJockey that uses program analysis to predict end of communication spurts and thereby accurately invokes fast dormancy without increasing signaling costs*.

*Work done while at Microsoft Research India.

MobiCom'12, August 22–26, 2012, Istanbul, Turkey.

The hypothesis behind our program analysis-based approach is that application developers typically invoke a unique set of functions in their code just before/after the end of communication spurts (e.g., cleanup activity such as close socket or flush buffers, update UI elements, etc.) If this hypothesis is true, by mining program execution traces offline, we should be able to automatically identify a set of program features (e.g., system or function calls) that only occur (or never occur) near the end of communication spurts. Since these features are inherent to the way the program operates, they should not be affected by the network dynamics, and closely track dynamic application communication behavior, the two issues that adversely affect a fixed idle timer-based approach.

The advantage of the above program analysis-based approach over a timer-based approach is two-fold. First, during the execution of the program, we can simply invoke fast dormancy as soon as the respective features are detected, thereby being even more aggressive in energy savings than the 3–5s idle timers. Second, despite being more aggressive in invoking fast dormancy, we are still assured that the application will not be communicating imminently (i.e., the idle-timer would anyway have fired, albeit after a 3–5s delay). Thus we avoid adversely impacting the network signaling load as compared to today.

RadioJockey operates in two modes, offline training and online application (Section 3). For offline training, we intercept system calls using either DETOUR [5] for windows applications or systrace for Android applications and collect program execution traces while the application is running in the background with no user interactions. We also simultaneously capture packet traces. From these traces, we extract a variety of features such as function calls, additional attributes like return values, arguments, history, stack depth, call stack, at the time of invoking the call. We mine these traces using the C5.0 decision tree-based classifier and classify periods as active (communication on-going) or inactive (end of communication spurt). A unique decision tree is learned for each application which is then used in the online phase for dynamically invoking fast dormancy during the execution of the application in the background. We have implemented the online runtime component of the RadioJockey system on Windows and show that it results in negligible runtime overhead (Section 4).

We evaluate RadioJockey using a variety of mobile smartphone as well as tablet and laptop applications for the android and windows platforms. The applications evaluated range from simple ones like gnotify, to complex applications like the Outlook email client (Section 5). We show that the rules mined have a high prediction accuracy and negligible false positive rates. A *surprising finding is that system call features alone are sufficient to identify high quality rules*, thereby allowing a simple, generic implementation to support a broad class of legacy, native, and managed applications. For over a dozen background applications, we show that RadioJockey is able to save 20-40% energy consumed by the cellular radio compared to a 3s idle timer while resulting in less than 2% increase in signaling costs for the network operator.

While application developers may be able to modify their code to invoke fast dormancy accurately at the end of each communication spurt [15], there are several advantages to using an automated approach like RadioJockey. First, most application developers are simply unaware of the energy characteristics of the cellular radio; RadioJockey allows these developers to focus on the design of their application while providing the energy savings of fast dormancy automatically. Second, based on the complexity of some of the rules that RadioJockey learns, we suspect that it is non-trivial for a developer to manually identify the end of all communication spurts for large complex applications such as Outlook. Finally, we believe

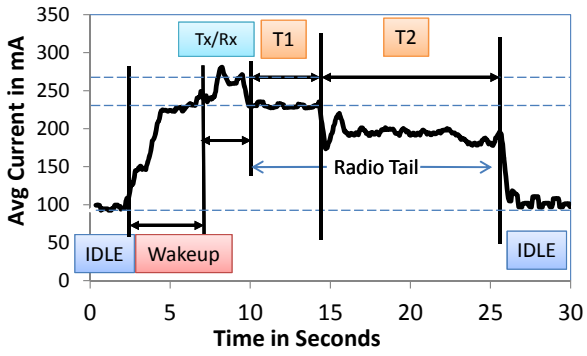


Figure 1: Energy consumption of the cellular interface

	Duration (s)	Energy(mJ)	#Signals
Ramp up IDLE to DCH	2	2	30
Ramp up PCH to DCH	0.5	0.5	12
Default Tail	20	15	2
5-sec Timer + FD	6	6	2
3-sec Timer + FD	4	4	2

Table 1: Cellular Radio Characteristics

that automated solutions like RadioJockey would allow platforms such as iPhone/Windows Phone to relax constraints imposed on background applications, secure in the knowledge that energy cost of *all background applications are automatically optimized*.

In summary, this paper makes the following contributions.

- We present a novel approach for accurately invoking fast dormancy based on mining program execution traces.
- We find that system call features alone are sufficient in identifying high quality rules, allowing generic fast dormancy support for broad class of legacy, native and managed applications.
- We present the design and implementation of RadioJockey that delivers 20-40% energy savings with negligible increase in operator signaling costs.

2. BACKGROUND AND MOTIVATION

2.1 Energy and Signaling Overhead

During normal usage, a cellular radio switches between different internal states called Radio Resource Control (RRC) states depending on the volume of network traffic. These state transitions are not instantaneous, and they incur both energy and signaling costs. This has a major implication on the energy consumed by background applications that typically perform short bursts of network activity for operations such as sending periodic heart beat messages, updating buddy status, checking for new emails, and pulling news feeds, etc. The energy consumption is mainly dominated by the switching overhead for short network transactions.

Figure 1 illustrates the power consumption characteristics of different states of a typical cellular radio during a short network session. Initially, when there is no network activity, the radio is in IDLE state, consuming very low power. It transitions to an active DCH state when there are one or more packets to be transmitted or received. The state transition (ramp up) usually takes up to 2 seconds since it requires authentication and a layer-2 connection to

be established, involving exchange of several signaling messages with different entities in the operator's network. The radio remains in DCH state while the network session is active. The radio is transitioned to a lower power state (IDLE or PCH), sometimes via another intermediate state (FACH), using an inactivity timer that is set by the operator. However, the timers are typically very large (for example, 20 seconds) resulting in significant energy cost for the radio-tail. Table 1 summarizes the energy and signaling overheads associated with different cellular states transitions.

To address the high energy drain due to long radio tails, many phone vendors use a feature called fast dormancy that enables quick transition to IDLE. The client device invokes fast dormancy using a shorter inactivity timer to cut down the radio tail. Fast dormancy is achieved by sending a specific signaling message called Signaling Connection Release Indication message (SCRI). In the early days, the mobiles simply used SCRI with cause value set to *unknown-cause* which resulted in releasing the RRC connection and moving the mobile to IDLE state. In some newer networks with support for 3GPP Release 8 Fast Dormancy (HSPA and LTE), the client sets the cause value to *PS Data session end* which allows the network to move the mobile to PCH state. PCH has slightly higher consumption than IDLE but requires fewer signaling messages to ramp-up from from PCH to DCH state (12 instead of 30 for IDLE) [9]. In either case, the signaling cost to transition to DCH is significant.

Note that *current best practices limit fast dormancy usage to background applications* [9]. This is because, when the user is interacting with the application, it is difficult to predict when the user may cause an interaction that results in new network traffic. In fact, iPhone OS v4.2 disabled fast dormancy usage when the screen backlight is on primarily because usage of fast dormancy for foreground applications in v4.1 resulted in significant increase in operator signaling costs [9]. Thus, in this paper, we restrict ourselves to applications that are running in the background while the screen is off, which typically constitutes a significant portion of mobile device usage.

While fast dormancy has been helpful in reducing energy consumption of background applications on mobile devices, existing approaches of employing aggressive timers can cause significant increase in the signaling load in the operator's network, even for networks that support PCH state.

2.2 Drawbacks of Timer based approach

We now describe two fundamental issues with fixed idle timer-based fast dormancy approaches, which can cause an unpredictable and significant increase in signaling load.

First, some applications are very chatty in the background, e.g., email clients such as Outlook and instant messaging clients like Lync and Skype. Figure 2 shows the CDF of inter-packet times when the client is running in the background. Notice that the inter-packet times span a large range of values with no distinctive knee. No matter what value is chosen for a fixed inactivity timer, packets can still arrive immediately after the timer has expired, causing the radio to be woken up immediately, thereby increasing the signaling overhead. RadioJockey operates at the granularity of sessions rather than packets. As we show in Section 5, our program analysis-based approach is able to distinguish between active and inactive sessions since it identifies/predicts the *e*nd of communication spurts.

Secondly, cellular network characteristics can change dramatically due to user mobility or changes in load characteristics. Packets may be delayed when there are sudden latency spikes. If aggressive idle timers are used, such scenarios can cause the signaling load to increase tremendously.

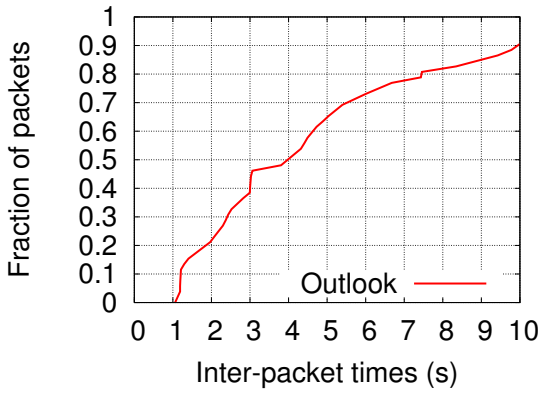


Figure 2: CDF of inter-packet times for Outlook application

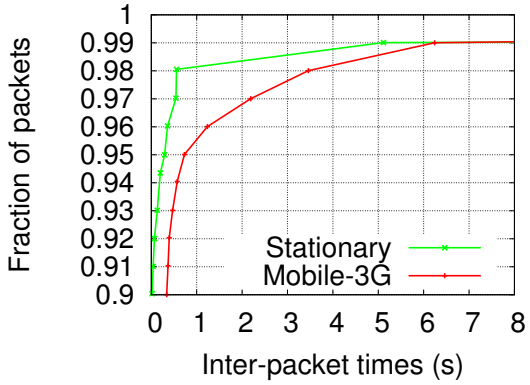


Figure 3: CDF of inter-packet times for Lync application on static and mobile scenarios

Figure 3 shows the CDF of packet inter-arrival distribution for Lync, an instant messenger application running in the background, for two cases: 1) when the node is stationary, and 2) when the node is mobile while attached to a 3G network. The 98 percentile of packet inter-arrival times increases from around 500ms for the stationary node to over 3s for the mobile node. Again, as we show in Section 5, since our program analysis-based approach is looking for rules identifying end of sessions, learning rules from the stationary node trace and applying the rules to the mobile scenario works well, which accords with the intuition that the program execution would remain unaffected by mobility.

In summary, inactivity timer based approaches are oblivious to whether there will a packet transmission in the near future. It is based on ad-hoc timer values, which is prone to errors. We take a different approach. We predict inactivity based on program behavior. Our prediction-based technique has two advantages: 1) it is able to shut down the radio as soon as the communication spurt ends, without waiting for an inactivity timer, and 2) makes it unlikely that there will be an immediate packet transmission, which would otherwise increase signaling costs.

3. RADIOJOCKEY DESIGN

The objective of RadioJockey is to learn *application-level* signatures that predict *network-level* behavior. In this section, we first describe our intuition behind this approach. We then provide rele-

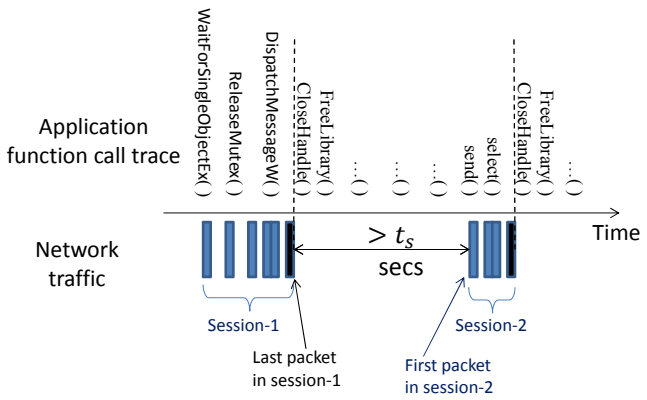


Figure 4: Defining end of sessions traces

vant definitions and describe RadioJockey's learning engine. Next, we describe the design of the run-time engine that implements rule-matching and the invocation of fast-dormancy. Finally, we describe several design decisions we made in the course of determining the specifics of the learning strategy.

3.1 Intuition

A program consists of one or more code paths (sequence of function calls) where each path is invoked in response to application logic and data. The various code paths can be identified by the function calls, stack traces, and system calls invoked as the program executes.

Often, for programs which perform sporadic background network tasks, a certain set of code paths lead to the application actively using the network, and a different set of code paths execute when the application stops using the network. The code paths include operations such as message processing, updating internal state, GUI window updates, memory allocation/deallocation, and closing of sockets/object handles. By identifying code paths that are correlated with the state of the program at the end of network sessions, and also during active network usage, we should be able to predict when the program is likely to stop communicating for a prolonged duration of time. Predicting such inactivity in advance allows us to determine when to invoke fast-dormancy and achieve energy savings while limiting the signaling overhead.

3.2 Definitions

We define an *active network session* as a time period over which the application sends or receives packets such that no two consecutive packets occur more than t_s seconds apart. t_s is the *end-of-session window* parameter. An *end-of-session* (EOS) is an event that occurs when the last packet of an active network session is seen. Therefore, t_s defines when EOS events occur in a given network trace that an application generates. Figure 4 depicts a traffic trace of an application, where packets are grouped into two active network sessions separated by more than t_s seconds.

The period t_s is chosen such that by invoking fast-dormancy immediately after the last packet in a session, there will be energy savings compared to a scenario where the radio remains in the active state between the two sessions. The goal of RadioJockey is therefore to predict whether the most recent packet transferred over the cellular interface happens to be the last packet in the current active network session, and if so invoke fast-dormancy.

If this prediction is correct (i.e., an idle timer based FD would have anyway fired for the same interval, albeit after a 3–5 second delay), we achieve the energy savings without any increase in signaling costs.

To check the feasibility of this idea, we collected function call traces that are executed at runtime for over a dozen background applications. We look at whether there are any unique function calls that are being called near an end of session. Visually inspecting did not seem promising as there were no obvious patterns in function calls called at an EOS event which we could manually discern. Consequently, we decided to use a machine learning approach towards discovering patterns that predict EOS.

3.3 Rule Learning

RadioJockey uses the C5.0 decision tree classifier as its principal learning tool. The algorithm builds trees using the concept of information gain. Starting at the root, it chooses the feature that splits the input data and reduces the entropy of the data set by the maximum amount. The final result is a tree in which branch points at higher levels of the tree correspond to attributes with greater predictive power. To avoid over-fitting, the algorithm includes a pruning step, wherein some branches in the tree are discarded while keeping the error with respect to the training data within a certain confidence threshold.

The input to the decision tree classifier consists of a set of labeled *data-items*. Each data-item consists of a set of key-value pairs, and the label assigned to it can either be "ACTIVE" or "EOS". The decision tree then outputs a set of boolean expressions, or *rules*, over these key-value pairs that indicate when a session is in an ACTIVE state, and when it has reached EOS. We now describe the method by which we generate the input data-items for training Radio-Jockey's classifier.

To profile an application using the classifier, we simultaneously collect raw system call traces and network traces while the application performs background tasks using the network. The required size of the traces varies depending on the complexity of the application. For a simple application, such as gnotify, whose sole purpose is to periodically poll an email server and determine if there is any new mail, the classifier may require only thirty minutes to an hour of data to learn accurate rules. On the other hand for a significantly more complex application such as the Outlook email client that performs multiple kinds of background network-related tasks, the classifier may need several hours of data to learn a potentially larger set of rules that characterize the application.

Figure 5 illustrates how we extract a set of training data-items from the system call traces and the packet traces. We use the network packet trace to divide the function call traces into active and inactive *segments* based on periods of network activity and inactivity. The period between any two consecutive packets is a segment. We create data-items from these segments using the following procedure.

Case 1 ($segment > t_s$)**:** If a segment is longer than t_s seconds, the system calls made from the time of the start of the segment up to a duration of t_w seconds form a data-item labeled EOS (middle data-item in Figure 5). The parameter t_w is called the *shutdown window* parameter and is less than t_s. We truncate every data-item to at most t_w seconds in length because we want to restrict our learning to features that lie within a short time of observing a packet. Without the truncation, the classifier may learn rules for EOS using features that occur well after observing a packet. Such a rule would

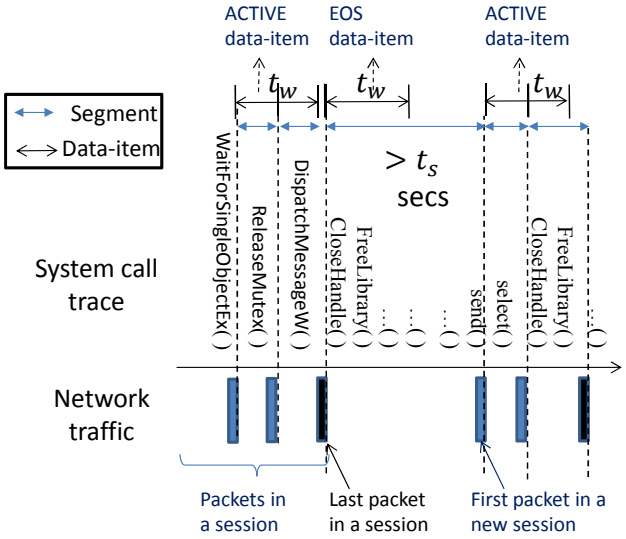
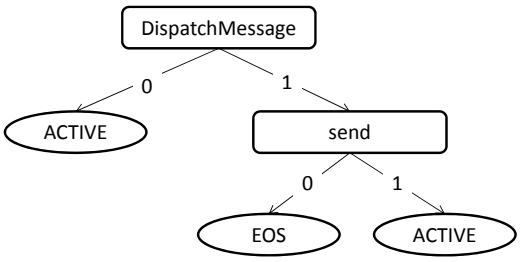


Figure 5: Generating data-items for the decision tree classifier.



(a) The decision tree learned for Gnotify

DISPATCHMESSAGE = 0 ⇒ ACTIVE (35)
DISPATCHMESSAGE = 1 AND SEND = 0 ⇒ EOS (24)
DISPATCHMESSAGE = 1 AND SEND = 1 ⇒ ACTIVE (1)

(b) The rules learned for Gnotify

Figure 6: Classifying Gnotify Behavior

be matched much after the actual end-of-session, and so the radio would be unnecessarily on from the end-of-session to the time of the match.

Case 2 ($t_w < segment < t_s$)**:** If a segment is shorter than t_s seconds but longer than t_w seconds apart, the system calls made from the first packet up to t_w seconds form an ACTIVE data-item (left data-item in Figure 5). This truncation, ensures that we solely learn rules involving system calls close to a network event.

Case 3 ($segment < t_w$)**:** Often, a large number of segments are formed by observing packets within very short time intervals. Due to inaccuracies in timestamping packets at capture-time, data-items created over very small durations may introduce a large amount of noise into the learning engine. Consequently, we concatenate as many such short consecutive segments as needed to create one ACTIVE data-item that is t_w seconds in duration (right data-item in Figure 5).

Thus every data-item contains system calls made over a period of t_w seconds. For each data-item, we extract binary features determining whether a certain system call existed in it or not. In other words, we use one binary variable for every system call. Its value for the data-item is 1 if the application made the call in this data-item, and 0 otherwise. In addition to these binary features, we also use a feature to indicate the previous data-item's state, that is, whether the data-item just prior to the current one was labeled ACTIVE or EOS. This parameter, which we call PREV_STATE, helps us capture temporal characteristics in the call trace. We found this particularly useful in capturing rules in complex applications, as we explain in Section 3.5.

Using this procedure, from the traces, we therefore generate a set of data-items labeled ACTIVE or EOS. From these, the classifier learns rules in the form of boolean expressions. We provide two *costs* to the learning algorithm. The cost of misclassifying an ACTIVE data-item as EOS has two components: first, we specify the energy expended in shutting-down the radio and then bringing it up again. Second, we specify the corresponding signaling overhead of mis-predicting an EOS. The cost of misclassifying an EOS data-item as ACTIVE involves purely the energy cost of keeping the radio up when it could have been put into fast-dormancy.

Figure 6a shows the tree learned for the gnotify application running on a Windows 7 system. The gnotify application periodically polls gmail and pops up a message when there is a new email (more elaborate results are shown in Section 5). The system call trace used to learn this tree was 5 minutes long. Every path from root to leaf yields one boolean expression. Consequently, the rules that the classifier learned for gnotify are listed in Figure 6b.

The numbers in parenthesis next to each rule are the number of data-items that contribute to learning that rule. They indicate that the first two rules have much higher confidence than the third rule. The DISPATCHMESSAGE system call dispatches an inter-thread message to a specified window procedure in the GUI thread, while the SEND call is used to send data out on a socket. Hence this set of rules indicate that, at run-time, in a data-item, if no DISPATCHMESSAGE is seen, the session is ACTIVE and therefore we should not invoke fast-dormancy. However, if DISPATCHMESSAGE is seen *and* SEND is *not* seen, the rules predict an EOS, and therefore we should invoke fast dormancy. If DISPATCHMESSAGE is seen *and* SEND is seen, then the session is still ACTIVE, and the radio should stay on.

It is easy to see why the SEND call appears in the rules since it directly results in network activity. Learning the DISPATCHMESSAGE call is, on the other hand, not as obvious. We believe that the classifier learns this call because gnotify, after polling the email server (i.e. at EOS), updates certain GUI artifacts (such as setting the last-checked-at message in the tool-tip of the application's icon in the system tray), which registers as a causality between EOS and DISPATCHMESSAGE.

Figure 7 shows a part of the tree learned for the Outlook email application running on a Windows 7 system. Outlook is a significantly more complex application than gnotify, and as a result, our learning engine learned approximately thirty rules of which the figure shows four prominent ones. An interesting observation is that the top-level feature in this tree is the previous state, or PREV_STATE: in Section 3.5 we explain why this is an important feature for complex applications. When PREV_STATE is EOS, the next session is predicted to be ACTIVE. On the other hand, if PREV_STATE is ACTIVE, if the LOCALFREE call is seen, the session remains ACTIVE. If LOCALFREE is not seen, but WSARECV is seen, the session is still ACTIVE. If WSARECV is not seen, the learning predicts that an end-of-session has been reached.

This tree shows the importance of learning different rules for different kinds of ACTIVE sessions that an application can have. Each path ending in an ACTIVE leaf node potentially represents a distinct background network behavior. It is important that our

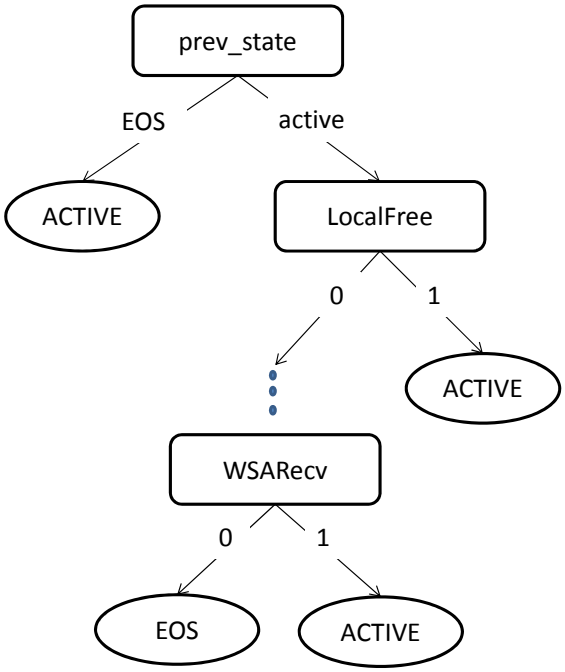


Figure 7: The decision tree learned for Outlook

learning technique capture each of these different behaviors automatically and completely.

3.4 Run-time

We now describe how our run-time engine uses the learned rules to decide when to shut down the network radio.

Whenever an application sends or receives a packet, say at time t, the run-time engine resets state and starts collecting system call traces as part of a new *session*. If the application sends or receives the next packet within the shutdown window t_w, we automatically determine that the current session (started by the first packet) is ACTIVE, and start collecting traces as part of the session started by the second packet. If, on the other hand, we see no network activity for the time window t_w since the first packet, the engine needs to decide whether the system calls in the current session predict EOS or not. If the set of system calls in the current session matches a rule for EOS learned in the rule-learning phase, thereby predicting that the network session has ended, the engine invokes fast dormancy. We elaborate further on this process while describing our implementation in Section 4. We repeat this process every time we see the application send or receive a packet.

3.5 Design Decisions

We now describe various decisions we made in the design of our data collection, learning and run-time engines.

Feature Selection: Initially, our learning engine used a significantly richer feature set that included application-level stack traces. We treated each call-site in the stack trace as a binary feature, just as we did with the system calls. We also included the return values to every system call and the system call as key-value pairs, as opposed to using just a binary variable to indicate whether a system call is present or absent in the data-item. We also tracked nested system calls (where a system call implementation calls another system call before returning to the caller). Surprisingly, we found that using only binary features for system calls was sufficient in learning accurate rules: adding the extra call stack information and the return values did little to increase the power of our learning. Tracking nested system calls made things worse because it introduced non-application-specific features (i.e., OS/library implementation features), which overwhelmed the application-specific features that are of primary interest. While counter-intuitive, this is a welcome result since tracking only the system calls that an application makes is a low-overhead task.

Use of previous state: While learning with only system calls worked well for simple applications, we found that the learning was noisy for more complex applications such as Outlook and Lync, even with stack traces. To address this, we added the binary feature PREV_STATE to every data-item, which tracks whether the previous data-item's label was ACTIVE or EOS. With this addition, our learning for complex applications such as Outlook improved dramatically. On closer observation, we found that these applications had different types of background network activity: while some sessions were long-lived and involved significant traffic, many others were very short and extremely periodic, such as application-level keep-alive messages being sent every 10 seconds. This behavior results in alternating data-items being in the ACTIVE and EOS states. Introducing the PREV_STATE parameter helped the classifier learn a different signature for such short and periodic sessions as compared to the longer, more heavyweight sessions.

Static vs Dynamic Learning: Our results indicate that using off-line, static learning on a per-user basis is very effective at learning accurate rules. We also found that our learning engine has very good turnaround time. For a complex application such as Outlook, using 6500 data-items involving 269 binary features, the classifier ran within 0.5 seconds on a desktop-grade machine. Since this is a fairly quick turnaround time in learning rules, we decided to implement a dynamic learning engine (as described in Section 4) in addition to the static learning engine. In case applications change their behavior over time, the dynamic learner running alongside the application will automatically generate new and updated rules.

4. IMPLEMENTATION

We have implemented the RadioJockey learning engine and run-time engine. Figure 8 describes the RadioJockey components at a high-level.

Learning engine. The learning engine comprises a call-tracing module, a packet-timings module, and the standard C5.0 decision tree classifier. The call-tracing module collects system call traces (on Android phones and tablets), and library call traces (on Windows tablets). System call traces are collected using STRACE. Library call traces are collected by injecting our RadioJockey DLL into the unmodified application's address space. The DLL intercepts calls to the 1634 core Win32 library functions (using DETOUR) and logs them (in-memory) before calling the intended function. Note that we do not need application source-code, nor do we need to re-write the application binary in either case. Using our STRACE and DETOUR approach we can support native as well as legacy applications.

The packet-timings module captures timing information for each packet sent or received by the application. We use LIBPCAP (on Android) and NETMON (on Windows). The key challenge we faced was associating a network packet to the running application process (without modifying the kernel). This was hard because LIBPCAP and NETMON do not support filtering packets by process ID. Our code performs the filtering in user-space. We link a packet to process based on the TCP port number. We get the port numbers opened by the application by periodically (every few milliseconds) snapshotting NETSTAT output. Since establishing a new TCP con-

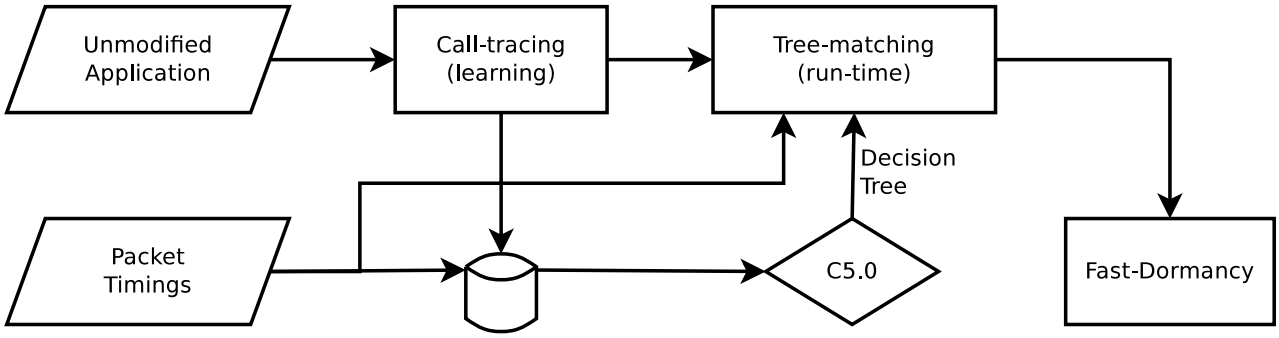


Figure 8: RadioJockey architecture

nection takes several round-trip-times (which are on the order of tens of milliseconds), we are still able to track short-lived sessions, although an in-kernel implementation would be more efficient.

The output of the call-tracing and packet-timing module are processed as described in Section 3.3 and fed into C5.0. The decision tree output by C5.0 is then fed into the run-time engine. Note that our implementation supports both offline and online learning. In offline learning the data used for learning may come from an earlier execution of the application. In online learning the data used for learning comes from the current application instance.

Run-time engine. The RadioJockey run-time engine comprises a tree-matching module, and a fast-dormancy module. The tree-matching module accepts a decision tree over a control channel, and classifies sessions as ACTIVE or EOS based on the logic described in Section 3.4. It shares much of its implementation with the call-tracing module since both use the same data. The decision tree can be updated at any time (and any number of times) without having to restart the application. When the tree-matcher predicts an EOS it informs the fast-dormancy module over a control channel.

The fast-dormancy module uses private APIs to invoke fast-dormancy on the particular platform (Android or Windows).

Overall our implementation consists of 2580 lines of C and C++ code (most of which is for the Windows DLL implementation), and 350 lines of Python scripts to process data-items and C5.0 output. Our RadioJockey DLL, which is injected into the running application's address space, is 1.4 MB in size and includes both the call-tracing and the tree-matching modules.

Avoiding mis-prediction cascades. We designed the RadioJockey run-time to avoid mis-prediction cascades. Specifically, we do not base the current prediction on the previous predicted value; PREV_STATE is always the *actual outcome* of the previous data-item. This also allows us to track the accuracy of our past predictions without incurring any additional overheads. For example, if we predict EOS and we see a packet before t_s has elapsed, we set PREV_STATE to ACTIVE to reflect the actual outcome, and log the false positive. If we don't see a packet within t_s of an ACTIVE prediction, we similarly set PREV_STATE to EOS and log the false negative. If the application behavior changes, or the application enters a phase that was not well-represented in the training data, this is manifested in our log as a string of mis-predictions, which the run-time engine detects. At that point the run-time (temporarily) stops making predictions, which gracefully falls back to existing behavior (of network operator defined idle timer based FD) with no additional signaling overheads.

The run-time engine resumes making predictions after some time. When resuming, it simply needs to wait for two consecutive packets (to set PREV_STATE); all other inputs come from the current data-item. Exponential back-off, where the run-time stops making predictions for exponentially longer intervals when the mis-prediction rate does not improve, ensures that the run-time closely tracks slight unexpected deviations in application behavior while minimizing overheads if application behavior changes significantly. When the exponential back-off timer exceeds a threshold, the run-time (optionally) switches to dynamic learning (Section 3.5) to re-learn the new application behavior.

App	Platform	Avg session(s)	Avg inter-session(s)	Complexity
Litestock	Win	0.51	14.83	Simple
Yahoo Messenger	Win	0.54	28.30	Simple
Seesmic	Win	0.69	35.66	Simple
Miranda IM	Win	0.31	49.38	Simple
Destroy Twitter	Win	1.99	48.77	Simple
Gmail Notify	Win	0.34	69.48	Simple
Desktop Changer	Win	6.99	63.46	Simple
GoChat Facebook	Android	0.18	26.22	Moderate
Google Talk	Win	0.28	27.39	Moderate
Tweetdeck	Win	0.37	24.71	Moderate
Twitdroid	Android	1.81	14.81	Moderate
Live mail	Win	11.42	71.11	Moderate
Lync	Win	1.03	55.73	Complex
Outlook	Win	0.86	18.53	Complex

Table 2: Applications and networks we evaluated RadioJockey on

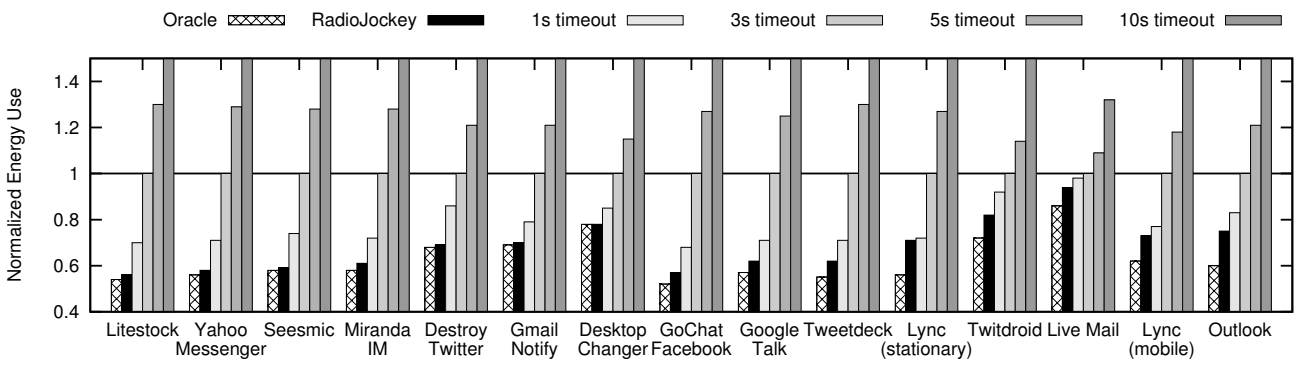
5. EVALUATION

We evaluate RadioJockey using both real-world experiments and trace-driven simulations across 14 applications on multiple platforms and on multiple network types. We find that our approach is robust to a wide range of application-level and network-level factors.

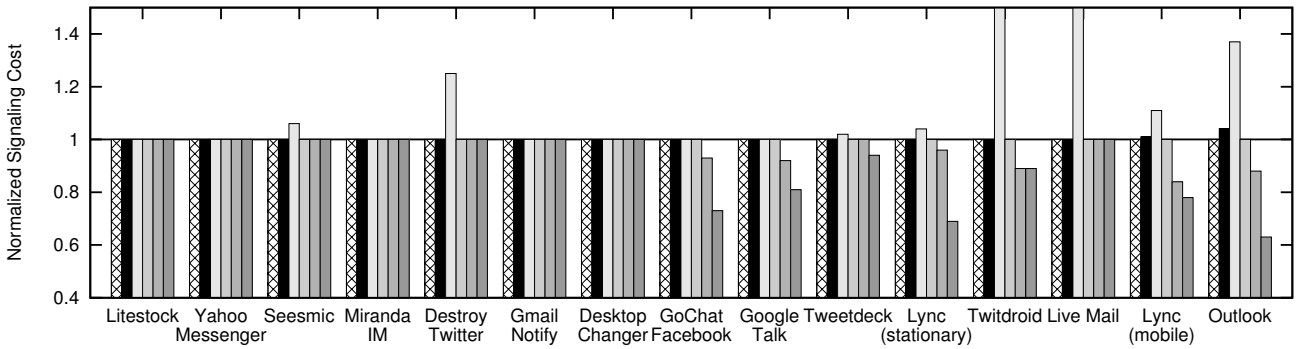
5.1 Macrobenchmarks

5.1.1 Energy Savings

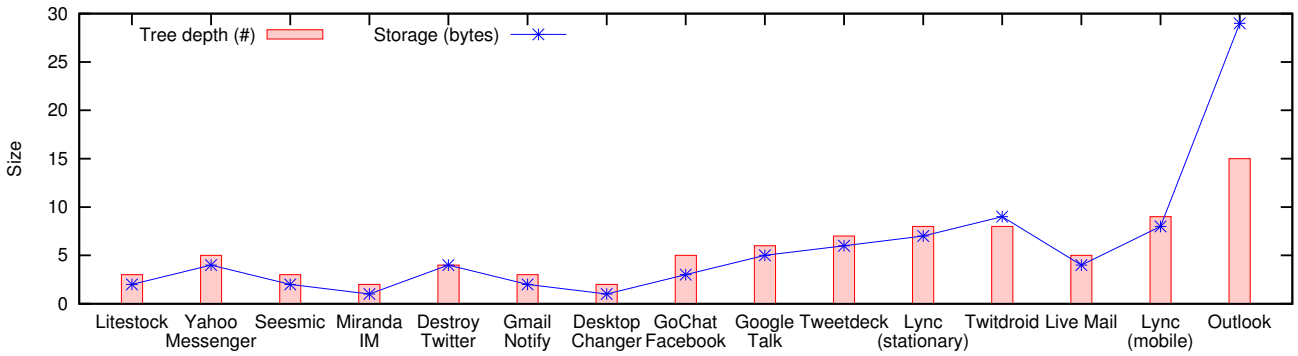
We ran the 14 applications listed in Table 2. Twitdroid and GoChat are Android applications, and the remaining are Windows applications. We captured call-traces and network-traces for each application. We performed a 10-fold cross-validation. That is, we partitioned the data into 10 parts, trained using 9 of them, and tested the classifier on the 10th; we repeated this 10 times for a different choice of the test partition. To compare with existing approaches, we simulate a *timeout-based* fast-dormancy approach for common timer values used today. The timeout-based approach invokes fast-dormancy when it sees a period of contiguous network inactivity for some fixed time.



(a) Energy savings



(b) Signaling overhead



(c) Tree complexity

Figure 9: RadioJockey energy savings and signaling overhead as compared to today, and complexity of the decision tree learned.

We additionally compare against an aggressive timeout-based approach that uses the same shutdown window (t_w) we use. Finally, to determine the maximal savings possible, we simulate an oracle that has perfect future knowledge about when a session ends so it can invoke fast-dormancy at the earliest possible time. We use energy, power, and signaling numbers measured on the Samsung Focus device mentioned earlier (Table 1) for computing these results. We disable RadioJockey's dynamic learning to evaluate the quality of rules we learn statically; enabling dynamic learning would serve only to improve our results.

Figure 9a plots the (normalized) median energy consumed for each approach and Figure 9b plots the (normalized) median signaling messages for each approach. All numbers are normalized to that of the 3 second *timeout-based approach*, which is the most aggressive (in terms of energy-savings) fast-dormancy strategy used by current smartphones [4]. The oracle, of course, consumes the least energy and has no added signaling overhead as compared to the 3s idle timer — this represents the optimal operating point.

For the simple applications, RadioJockey performance is nearly identical to that of the oracle, saving between 44%–22% of energy as compared to today without any additional signaling overheads. For these applications, RadioJockey correctly predicted every EOS (i.e., 0% false-negative) and did not mis-predict any ACTIVE session (i.e., 0% false-positive); the slight energy difference compared to the oracle is due to the small shutdown window that RadioJockey needs to wait before entering fast-dormancy.

While an aggressive timeout-based approach achieves only slightly lower energy savings, it can have significantly higher signaling

cost (between 6%–56% more than today across all applications). The reason for this, as mentioned, is that such an approach will be highly sensitive to application and network factors, often mis-predicting an EOS if an application stalls briefly or packets are delayed. As mentioned, each mis-prediction (false-positive) incurs an additional signaling cost of 32 messages (for 3G or for LTE) as compared to today. More conservative timer based approaches have lower signaling overhead (since they enter fast dormancy less often than today), but correspondingly consume a lot more energy than today.

For the moderate complexity applications, RadioJockey still outperforms all timeout based approaches, and achieves 42%–90% of the oracle's energy savings (i.e., 6%–43% savings compared to today) without any added signaling overhead. This is attributed to RadioJockey opting to minimize false-positives (i.e., not incurring added signaling overhead) at the expense of false-negatives (i.e., not saving as much energy as optimally possible) as mentioned earlier. The performance of timeout based approaches is qualitatively similar to before — higher signaling overheads for aggressive timeouts, and lower energy savings for conservative timeouts.

For complex applications, RadioJockey manages to achieve around 65% of the oracle's energy savings (25% compared to today) with only 1%–4% signaling overhead. The timeout based approaches, as before, do not achieve comparable energy savings and signaling overheads.

Next we focus on the complexity of the decision tree learned by RadioJockey (Figure 9c). The figure plots the depth of the tree and the storage overhead (in bytes). The tree depth tracks the number of boolean conditions that need to be checked at run-time. The storage tracks the memory required to store these booleans. As is evident from the graph, as application complexity grows, so does the complexity of the tree. At an absolute level though, run-time processing and storage requirements are minimal (requiring only 15 comparisons and 29 bytes of storage even for applications as complex as Outlook).

5.1.2 Run-time Engine Evaluation

We now focus on experimentally evaluating the RadioJockey run-time engine. The run-time operation needs to be highly performant in order to issue fast dormancy commands to the cellular interface in a timely manner. We ran 4 separate applications of different complexity. Each application ran in isolation for around 2 hours in the background on a Windows 7 based tablet device. The decision tree used for classification is built offline for each application. We measured how often the RadioJockey run-time mis-predicted EOS (false positives), and what fraction of EOS events are predicted accurately (compliment of false negatives). Energy spent by the network interface and signaling overhead is estimated offline using network capture trace and timestamps of EOS predictions when fast dormancy commands are invoked. Each mis-prediction by the run-time engine leads to an unnecessary state transition thereby spending one additional ramp-up and tail energy cost as well as increased signaling messages. Lower accuracy implies reduced energy savings since RadioJockey run-time falls back to using timeouts to invoke fast dormancy for EOS events that are not predicted.

Figure 10 shows percentage values for false negatives, false positives, energy savings, and signaling overhead compared to a default 3 second timer based approach. We find that false negatives to be under 3% for the four applications we tested, and as a result we obtain energy savings of 20%–30%. However, we find that signaling overhead is marginally higher (3%–6%) compared to offline analysis owing to increased mis-predictions in run-time. This is due to

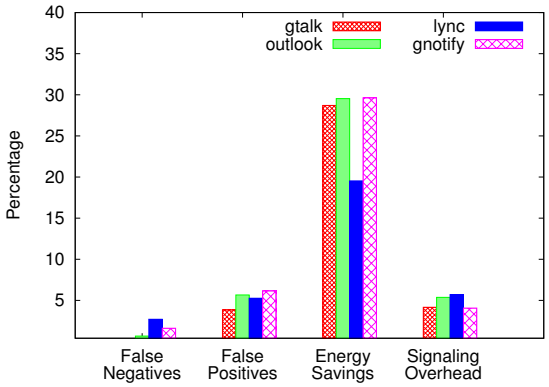


Figure 10: Runtime performance with respect to 3s timeout scheme for four applications.

inaccuracies in our the packet-timings module, which in our current prototype, as mentioned, is implemented in user-space and needs to make heavy use of netstat to link packets to processes. As a result, it experiences scheduler-related delays and jitters. An in-kernel implementation that has direct access to kernel data structures and is not affected by the process scheduler would significantly improve the accuracy of packet timestamps. That being said, our current user-space approach still delivers most of the energy savings with low signaling overheads in practice.

5.1.3 Phone experiments

We next measure the energy saved by RadioJockey on a Samsung Focus Windows phone connected to a 3G network. In order to measure only radio related energy costs, we turn off the screen and measure the base power consumed by the phone using a Monsoon power monitor. Next with the screen still off, we run a representative application (gnotify) on a device tethered to the phone (so all network traffic uses the phone's radio) while measuring the power consumed. We then run the application with the RadioJockey run-time engine enabled, which sends fast-dormancy commands to the fast-dormancy module running on the phone. Finally, for comparison, we measure the power for a simple timeout based fast-dormancy approach where our fast-dormancy module on the phone fires after the configured period of inactivity. We integrate the power over time to compute the energy consumption.

For the 3 second timeout-based approach, the phone consumed 94.86 J of energy, whereas using the RadioJockey engine, the phone consumed only 72.2 J. RadioJockey therefore saves 24% energy over the aggressive 3 second timeout-based approach, which is in line with our simulation results. This demonstrates the expected energy savings promised by RadioJockey can actually be attained in real-life scenarios.

5.1.4 Robustness

Network. To demonstrate the robustness of the decision tree learned by RadioJockey, we experiment by running the learning-engine on a static device, and use the tree to classify the application on mobile device. Figure 11 plots the numbers for the Lync application. Even though the network characteristics (latency, packet loss, bandwidth) for the two scenarios are very different, RadioJockey manages to extract 28% energy savings. This was identical to training and testing on the same network condition. The ability to use traces collected across different network types demonstrates the robustness of RadioJockey to network characteristics, allowing

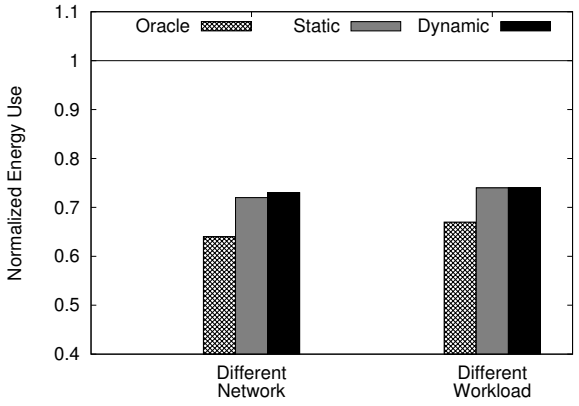


Figure 11: Robustness of the decision tree learned by RadioJockey to changing network and workload conditions.

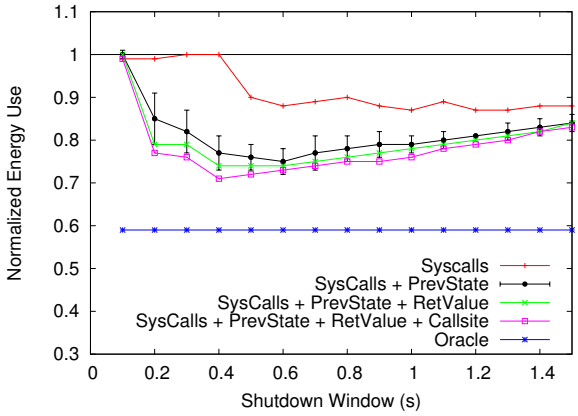


Figure 12: Selecting features and shutdown window size.

Applications	Energy Savings (%)	Signaling Overhead (%)
Outlook	24.03	4.47
GTalk	24.07	4.57
Lync	24.14	0
All	22.8	6.96

Table 3: RadioJockey with multiple applications.

for scenarios where the phone or tablet platform provider may perform the RadioJockey offline learning and provide prediction rules to the users, so that users get some baseline energy savings regardless of their operating scenario.

Workload. To further explore the robustness of the decision tree, we experiment by learning on one workload, and testing on a different workload. We use the Lync application where user one has a small number (9) of friends, while user two has many more (25). We find that RadioJockey saves 26% energy in both cases, further demonstrating the robust nature of the decision trees with respect to workloads. This suggests that RadioJockey traces can additionally be shared across users further reducing the learning burden for individual users.

5.1.5 Multiple applications

So far we have measured each application in isolation. In practice, multiple background applications may be running at any given time, and the radio can go into fast-dormancy only if none of the applications is actively communicating. This naturally reduces the energy savings. We simulate different interleaving of three applications – Outlook, Lync and GTalk. For this we use the decision trees learned earlier, and enter fast dormancy when neither application is classified to be in an ACTIVE session.

Where RadioJockey saves 24% energy for applications in isolation compared to the 3 second timeout approach, for the interleaved trace, RadioJockey saves on average 23%. The signaling overhead only increases marginally to 7%, whereas individually the signaling overhead is below 5% for each application in isolation. The small increase in signaling overhead is due to the interference between different applications. Although we are predicting EOS events accurately, RadioJockey does not predict the start of a session (SOS). As a result, when EOS is predicted for one application leading to fast dormancy, an SOS event for another application can potentially result in waking up the radio immediately resulting in a false positive. However, the data suggests that active sessions of multiple background applications do not overlap often. Thus RadioJockey still manages significant energy savings even while running multiple background applications together with marginal increase in signaling overhead.

5.2 Parameter Selection

We next focus on our choice of features to base RadioJockey's learning algorithm on. Figure 12 performs a parameter sweep for t_w (shutdown window) on the x-axis for Outlook, and plots the energy savings for the choice of application features. Signaling overheads (not shown) are low throughout.

Note first that using only system call information without PREV_STATE performs quite poorly, saving at best only 13% energy (as compared to the oracle saving 41%). The reason, as we have noted, is that the tree did not distinguish between long- and short- ACTIVE session. To better handle this temporal state, we add the PREV_STATE feature.

Adding previous state significantly improves energy savings (increasing it to 25% in the best case). Beyond this, small gains are to be had by including return values, and callsite and stack trace information (adding a further 1% and 4% respectively), however given the significant overheads in acquiring a stack trace at every intercepted function call, we decide to forgo this small boost.

Interestingly we found that for all the applications we tested (and all sets of features we tested), the energy savings curve follows the same pattern — energy savings linearly increase monotonically as the shutdown window size decreases, but only up to a point. Beyond that the energy savings decrease monotonically as the window size decreases further still. This is explained by the fact that when the shutdown window size is large, all EOS are predicted correctly and false-positives are rare, and energy savings is a linear function of how long RadioJockey must wait to make a prediction (i.e., shutdown window size t_w). Beyond the critical shutdown window size, false negatives increase as there is too little data to predict every EOS, which results in lower energy savings.

Based on empirical measurements across different networks, and different workloads, this critical point appears to be stable for a given application (but different for different applications). This suggests that this is an application-dependent quantity that can be learned by the platform during application evaluation time, and used by all users.

6. DISCUSSION

6.1 Network Features

RadioJockey employed features from application execution traces to infer rules that predict an EOS events effectively. We also evaluated whether network features alone can be used for predicting the same. We used features such as TCP flags, HTTP content length, packet payload length, and hash of first few bytes of the packet. The intuition for using hash value is that many applications use custom protocols for communication and most often the first few bytes of the packet carry header related information.

For applications which tear-down network connections at the end of a session, we are able to find good rules that predict EOS based on TCP flags alone. These included simple background applications such as Desktop Changer, FeedReeder, and GNotify. However, for many applications that keep long standing active TCP connections for communicating with the server, we could not find high accuracy rules. Another major drawback of this approach is that it cannot be employed on applications that use encryption.

6.2 Temporal Rule Mining

Prior to using decision tree based classification technique in RadioJockey, we employed temporal rule mining to learn application specific rules from system call traces and network traces. The approach inferred rules of the form $A \rightarrow B \rightarrow ... \rightarrow C$ where A, B and C are predicates that frequently occur in a sequence in the traces, however not necessarily next to each other. A predicate can either be a function name with optional return value parameter or an EOS event (obtained from network trace). We inferred rules which have EOS as one of the predicates. To limit the search space, we only find rules where the first and last predicate occur within a certain time duration. We selected a subset of rules that are able to capture most of the EOS packets in the training set with minimal false positives. In addition, we preferred rules which end with an EOS predicate. At runtime, whenever a series of predicates in a rule are observed, fast dormancy command is issued immediately after the last predicate. As a result temporal rule based approach can potentially save more energy compared to classification based approach that uses system calls captured after the occurrence of a packet, thus taking marginally longer time to invoke fast dormancy.

For most simple applications a small set of temporal rules with at most 4 predicates is sufficient to predict EOS events with high accuracy. However, inferring temporal rules for complex applications like Lync and Outlook, which have a lot of system calls, turned out to be time consuming as the search space blows up exponentially with number of predicates. This was one of the reasons for using classification technique in RadioJockey.

6.3 Foreground Applications

RadioJockey is currently designed to invoke fast dormancy only for background applications during periods when there are no active user interaction. Predicting EOS events for foreground applications turns out to be challenging since user interactions can trigger network communications at any point in time. Accordingly, RadioJockey run-time employs a simple heuristic — enable run-time only when the screen is idle. This ensures that fast dormancy calls for background applications do not interfere with foreground applications. Although this approach may seem very conservative, mobile phone usage is typically dominated by long periods of user inactivity during which background syncing activities are carried out. We plan to focus on foreground applications as a part of future work.

7. RELATED WORK

Understanding cellular radio energy characteristics has received increasing attention in recent years. In particular, the energy cost of the "tail" [2, 18] of the 2G/3G cellular radio, where the radio remains in a high power state for a large inactive duration (determined by the operator) after the end of each data communication spurt, has received significant attention. We describe below related work that has focused on this aspect of the cellular radio.

Measurements and Modeling. Several papers [12, 11, 14] have used active measurements to characterize the cellular radio state transition parameters of different operators. These measurements indicate that default parameter values for the cellular radio energy tail can vary across operators and be as high as 20 seconds, resulting in significant energy drain on smartphones. Falaki et al. [3] analyze smartphone traffic from a large number of users and show that 95% of inter-packet arrival times lie within 4.5 seconds, thereby arguing for a shorter idle timer value for 3G radios.

Analytic models have been proposed for computing the energy consumption of 3G radios [10, 19]. Authors in [19] compute the energy cost of using different inactive timers for 3G radios by modeling traffic and 3G radio characteristics. Pathak et al. [10] use system call tracing to perform fine-grained modeling of energy consumption, including modeling the various states of the 3G radio, for accurate energy accounting of applications on mobile smart phones.

Optimizing for a given tail. Several papers [2, 6, 8, 17, 18], have leveraged pre-fetching and/or delayed scheduling in order to amortize the energy cost of the cellular radio tail. Tailender [2] uses a combination of delayed transfer and prefetching of search query results in order to reduce energy. Cool-Tether [18] takes the energy cost of the tail into account for deciding the number of phones to be combined for tethering. Bartendr [17] prefetches data at good signal strengths for streaming and background transfer applications in order to save energy.

Optimizing the tail. Modern smart phones use idle timer values that range from 3 to 10s for invoking fast dormancy [4], a considerably smaller duration than the typical 12-20s of default tail durations configured by operators [14]. However, these idle timer values are currently chosen in an ad hoc manner, balancing operator sensitivities to signaling costs of low idle timer values with the needs of reducing smartphone energy consumption. Several papers have also proposed to dynamically choose idle timer values based on traffic characteristics instead of static idle timer values used currently [7, 13, 19]. However, as we show in Section 2, it is not always possible to choose an aggressive idle timer value that accurately predicts the end of a communication spurt, given network characteristics and user mobility can cause significant variation in inter-packet arrival times. Thus, these approaches can result in large increases in signaling overhead.

ARO [16] is a tool that collects various profiling data when an application is running (e.g., packet traces, user interacation, etc.) and then analyzes the data offline in order to shed light on the impact of various smartphone application features and their respective radio energy costs. Thus, ARO helps provide developer with sufficient cross-layer information (TCP, radio states, etc.) and the energy cost of different functions of the application, thereby allowing the developer to redesign his code in a more energy efficient manner.

Perhaps closest to RadioJockey is TOP [15] where applications are modified to leverage fast dormancy in order to save energy. Applications in TOP actively inform the network when their communication spurt is finished. Thus, the cellular radio is able to quickly go into a low energy state. However, TOP requires application de-

velopers to be aware of the cellular energy tail characteristics and incorporate this into their code design. RadioJockey has the same goal of invoking fast dormancy after the end of a communication spurt but without requiring any support from the application developer. By mining program execution traces, RadioJockey is able to infer rules for end of communication spurts and can thus automatically invoke fast dormancy at the correct times without any change to application code. The complexity of some of the rules that we learn for identifying the end of a communication spurt argues for an approach such as RadioJockey since an application developer may not always be able to identify these rules, especially for large and complex applications.

8. CONCLUSION

Many applications such as email clients, instant messenger, news readers, etc. run in the background on a mobile device and perform network activity for synchronizing state information. However, the energy cost of running these applications in the background on a cellular radio interface can be significant enough that many smartphone platforms prohibit background application or severely restrict them. Recent smartphone models cut down the energy cost by implementing a feature called Fast Dormancy (FD), which forces the radio to go to a low energy state based on a short inactivity timer. However, such idle timer-based approach face two drawbacks: some applications have a large variance in their packet inter-arrival distributioin and second variation in network conditions due to mobility can also change the packet inter-arrival distribution.

In contrast, our system called RadioJockey analyzes program execution traces and mines rules for identifying end of communication spurts. We show that our approach is able to save 20-40% energy savings compared to an idle timer approach for a large class of applications, is much more robust to variations in network conditions, and achieves savings with negligible increase in signaling load for the network operator.

9. REFERENCES

[1] How smartphones are bogging down some wireless carriers. http://bit.ly/Nc0XgL.

[2] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the Internet Measurement Conference (IMC)* (Chicago, IL, Nov. 2009), ACM.

[3] FALAKI, H., LYMBEROPOULOS, D., MAHAJAN, R., KANDULA, S., AND ESTRIN, D. A first look at traffic on smartphones. In *Proceedings of the Internet Measurement Conference (IMC)* (Melbourne, Australia, Nov. 2010), ACM.

[4] HUAWEI. Behavior analysis of smartphones. http://bit.ly/Nc0US7.

[5] HUNT, G., AND BRUBACHER, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (Seattle, WA, July 1999).

[6] LAGAR-CAVILLA, H. A., JOSHI, K., VARSHAVSKY, A., BICKFORD, J., AND PARRA, D. Traffic backfilling: subsidizing lunch for delay-tolerant applications in UMTS networks. In *Proceedings of the 3rd Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld)* (Cascais, Portugal, Oct. 2011).

[7] LIERS, F., AND MITSCHELE-THIEL, A. UMTS data capacity improvements employing dynamic RRC timeouts. In *Proceedings of the 16th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)* (Berlin, Germany, Sept. 2005).

[8] LIU, H., ZHANG, Y., AND ZHOU, Y. TailTheft: Leveraging the wasted time for saving energy in cellular communications. In *Proceedings of the 6th ACM International Workshop on Mobility in the Evolving Internet Architecture (MobiArch)* (Washington, D.C., June 2011).

[9] NOKIA SIEMENS NETWORKS. Understanding Smartphone Behavior in the Network, Nov. 2010. http://bit.ly/OUEkl1.

[10] PATHAK, A., HU, Y., ZHANG, M., BAHL, P., AND WANG, Y. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)* (Salzburg, Austria, Apr. 2011), ACM.

[11] PERALA, P., BARBUZZI, A., BOGGIA, G., AND PENTIKOUSIS, K. Theory and practice of RRC state transitions in UMTS networks. In *Proceedings of the 5th IEEE Broadband Wireless Access Workshop (BWA)* (Anaheim, CA, 2009).

[12] PERALA, P., RICCCIATO, F., AND BOGGIA, G. Discovering parameter settings in 3G networks via active measurements. *IEEE communication letters 12*, 10 (2008).

[13] PUUSTINEN, I., AND NURMINEN, J. The effect of unwanted Internet traffic on cellular phone energy consumption. In *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (Paris, France, Feb. 2011).

[14] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Characterizing Radio Resourcde Allocation for 3G Networks. In *Proceedings of the Internet Measurement Conference (IMC)* (Melbourne, Australia, Nov. 2010).

[15] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. TOP: Tail Optimization Protocol for Cellular Radio Resource Alocation. In *Proceedings of the 18th International Conference on Network Protocols (ICNP)* (Kyoto, Japan, Oct. 2010).

[16] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (Washington, D.C., June 2011), ACM.

[17] SCHULMAN, A., NAVDA, V., RAMJEE, R., SPRING, N., DESHPANDE, P., GRUNEWALD, C., JAIN, K., AND PADMANABHAN, V. N. Bartendr: A Practical Approach to Energy-aware Cellular Data Scheduling. In *Proceedings of the 16th International Conference on Mobile Computing and Networking (MobiCom)* (Chicago, IL, Sept. 2010).

[18] SHARMA, A., NAVDA, V., RAMJEE, R., PADMANABHAN, V., AND BELDING, E. Cool-Tether: Energy Efficient On-the-fly WiFi Hot-spots using Mobile Phones. In *Proceedings of the 5th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (Rome, Italy, Dec. 2009).

[19] YEH, J.-H., CHEN, J.-C., AND LEE., C.-C. Comparative Analysis of Energy-Saving Techniques in 3GPP and 3GPP2 Systems. *IEEE Transactions on Vehicular Technology (TVT) 58*, 1 (2009).