SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2017, Vol. 60 012106:1–012106:16
doi: 10.1007/s11432-015-1026-y

DelayDroid: an instrumented approach to reducing tail-time energy of Android apps

Gang HUANG[1]*, Huaqian CAI[1], Maciej SWIECH[2], Ying ZHANG[1], Xuanzhe LIU[1] & Peter DINDA[2]

[1]*School of Electronics Engineering and Computer Science, Peking University, Bejing* 100871*, China;*
[2]*Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL* 60208*, USA*

Received May 20, 2016; accepted July 8, 2016; published online November 17, 2016

Abstract Mobile devices with 3G/4G networking often waste energy in the so-called "tail time" during which the radio is kept on even though no communication is occurring. Prior work has proposed policies to reduce this energy waste by batching network requests. However, this work is challenging to apply in practice due to a lack of mechanisms. In response, we have developed DelayDroid, a framework that allows a developer to add the needed policy to existing, unmodified Android applications (apps) with no human effort as well as no SDK/OS changes. This allows such prior work (as well as our own policies) to be readily deployed and evaluated. The DelayDroid compile-time uses static analysis and bytecode refactoring to identify method calls that send network requests and modify such calls to detour them to the DelayDroid run-time. The run-time then applies a policy to batch them, avoiding the tail time energy waste. DelayDroid also includes a cross-app communication mechanism that supports policies that optimize across multiple apps running together, and we propose a policy that does so. We evaluated the correctness and universality of the DelayDroid mechanisms on 14 popular Android apps chosen from the Google App Store. To evaluate our proposed policy, we studied three DelayDroid-enabled apps (weather forecasting, email client, and news client) running together, finding that the DelayDroid mechanisms combined with our policy can reduce 3G/4G tail time energy waste by 36%.

Keywords refactor, Android, optimization, energy, network scheduling

Citation Huang G, Cai H Q, Swiech M, et al. DelayDroid: an instrumented approach to reducing tail-time energy of Android apps. Sci China Inf Sci, 2017, 60(1): 012106, doi: 10.1007/s11432-015-1026-y

1 Introduction

Smartphone apps typically use mobile networks to carry data traffic. When data is to be transferred to a base station, a phone's mobile radio (e.g., 3G or 4G radio) operates in a high power mode. When no data is to be transferred, the radio will first switch to a so-called tail mode that consumes less energy, and later will switch to an idle mode that consumes even less energy. The intermediate tail mode is a compromise between energy consumption and fast response to new data transfer requests. However, it can still lead to a waste of energy when no data is transferred while in tail mode [1].

Reducing such energy consumption can prolong the battery life of a smartphone. One approach is to batch lower priority or delay-tolerant network requests sent by apps, reducing the occurrence of tail mode.

* Corresponding author (email: hg@pku.edu.cn)

Prior research efforts [1–4] have proposed request delaying or prefetching algorithms and policies to batch network requests. However, applying such policies to existing apps, both to deploy the policies and to evaluate them, is a challenging problem. We describe the design and implementation of a framework, DelayDroid, which supports such deployments, and use it to evaluate our own policy. DelayDroid allows policy deployment on existing, unmodified apps with no human effort.

Approaches to applying algorithms and policies to existing apps can be classified according to the layer at which they operate: application layer, OS layer, and framework layer.

In an application layer approach, developers implement their algorithms/policies within individual apps by hand. While this leverages the best available information about when batching is acceptable, it has several limitations. First, it can only batch the requests of the individual app, but the radio is shared by all the apps on the phone. Second, it limits the opportunities to batch requests. Within a single app, it may be the case that two requests cannot be batched because there is a dependency from one to the next (e.g., the second request needs the response of the first request). However, it may be possible to batch the first request with the request of an unrelated app. More generally, the larger the scope of requests we consider, the greater the number of potential groupings of requests into batches.

In an OS layer approach, the system calls to support request handling are redesigned. Because all requests flow through the OS, the OS can batch requests globally, quite unlike the application layer approach. Furthermore, minimal application changes may be needed, as the heavy lifting happens under the stable system call interface. However, this approach also has limitations. First, redesigning the OS to support request batching of massive apps is challenging, as is the acceptance of such changes. Second, because the system call interface discards considerable semantic information, the OS's decision-making operates at an information disadvantage compared to the application layer approach. Finally, specialization for specific apps or groups of apps is impossible.

In a framework layer approach, libraries are updated to provide new APIs that allow apps to specify request batching. For example, iOS provides APIs for developers to indicate delay-tolerant network requests and batch them. Compared to the application layer and OS layer approaches, the framework layer approach is lighter weight, incrementally deployable, and capable of integrating batching across all the apps that make use of it. However, it does require considerable human effort on the part of developers. Developers must rewrite or modify their apps to use the new APIs. This is particularly challenging for large, complex apps.

DelayDroid is a framework layer approach, but it leverages compile-time techniques to adapt apps to the framework instead of requiring the developer to do it. Given a set of apps, DelayDroid first analyzes their bytecode to find the network-relevant classes, those that make requests. It then refactors these classes to introduce calls to its run-time. DelayDroid's compile-time operations are done with no developer effort. The run-time then schedules these requests according to some desired policy. Instead of updating the libraries, DelayDroid automatically analyzes apps and generates network-request-batched apps. Our work makes the following contributions:

- We introduce an automated method of inserting a network request batching mechanism into groups of complex apps at the framework layer. By batching network requests, we are able to reduce app tail time energy consumption.

- We describe the design, implementation, and evaluation of an instance of this approach, DelayDroid, showing how it operates on existing, unmodified Android apps. Our evaluation is based on 14 apps chosen from the Google App Store.

- We describe the design, implementation, and evaluation of a new policy for request batching. Using DelayDroid, we apply our policy to a group of three existing apps, and find it is able to reduce the energy wasted in tail times with an average reduction of 36%.

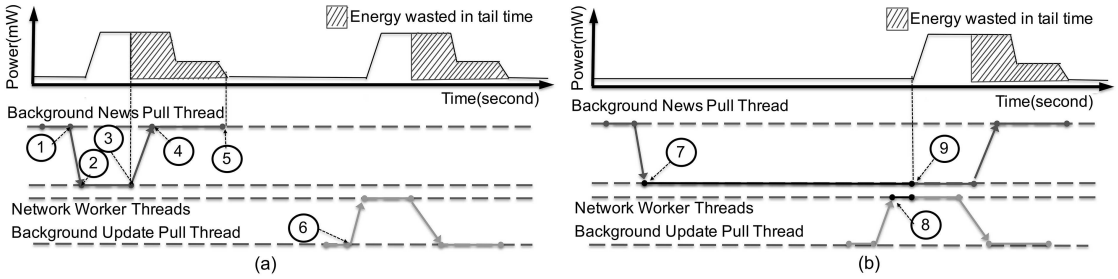

Figure 1 Motivating example. (a) Original network trace; (b) batched network trace.

2 Background and motivation

Today, 3G and 4G are the most widely used mobile networks. We now describe power management in 3G/4G radios and give an example of how careful delaying and batching of requests can lead to reduced energy consumption by reducing the occurrence of tail time.

2.1 3G/4G power management

3GPP (Third Generation Partnership Project[1)]) and 4GPP[2)] are the most common mobile communication standards today. They have similar state machines for the radio resource control (RRC). The state machine consists of three states: IDLE, CELL_DCH (Dedicated Channel) and CELL_FACH (Forward Access Channel). In the IDLE state, the radio is in a low power state. When the network is activated, the radio switches to the CELL_DCH state, which provides high bandwidth and low latency transmission at the cost of high power consumption. In the CELL_DCH state, after the network is idle for a few seconds, the radio switches to the CELL_FACH state, which has lower bandwidth and higher latency, but consumes only half of the power of the CELL_DCH state. In the CELL_FACH state, after the network is idle for a few more seconds, the radio switches to the low power IDLE state. The duration of inactivity before the radio demotes from the CELL_DCH state to the IDLE state is defined as the tail time. Network operators trade off between network performance and energy consumption [5], typically configuring the inactivity timeout to be several seconds long.

This mechanism can not only reduce the latency incurred in the promotion from the idle state to the high power state, but also reduce the signaling overhead incurred in the state transition process due to the allocation and release of the communication channel itself (which is released in the IDLE statement and must afterwards be reacquired). However, in all cases, a transition sequence of CELL_DCH→CELL_FACH→IDLE wastes energy due both to the time spent in CELL_FACH (which was completely unneeded) and the extra time spent in CELL_DCH.

2.2 Motivating example

Figure 1(a) shows a timeline in which we send two network requests with an interval of 20 s between them. The upper half of Figure 1(a) is the power consumption of the radio. The dotted lines in the bottom half denote threads, and the solid arrows denote control flow. Control flow starts with the "background news pull" thread waking up the "network worker" thread in the network thread pool ①. Then the "network worker" thread issues a network call to send network requests ②. In response, the radio promotes to CELL_DCH, the high power state. Finally, after receiving the response ③, the "network worker" thread returns the response to the "background news pull" thread ④. Now that the first request is finished ③, the radio remains in each of the CELL_DCH (high power) and CELL_FACH states for a few seconds. After that, the radio demotes to the IDLE state ⑤. The interval between ③ and ⑤, the time during which the radio could have been in IDLE is called tail. The energy wasted in the tail is marked in

1) Third generation partnership project (3gpp). http://www.3gpp.org.
2) Fourth generation partnership project (4gpp). http://www.4gpp-project.net/.

backslashes in Figure 1. After some time, the "background update pull" thread wakes up ⑥, and the process repeats, as does the wasted energy of the tail.

Now consider that it may be appropriate to delay a "background news pull" task, at least at the granularities (10 s) involved in this example. Such delays give us the opportunity to reduce the occurrence of the tails, as well as to batch work together for when the radio is active. Figure 1(b) illustrates the process of batching two network requests in our example. As before, the "background news pull" thread wakes up the "network worker" thread in the network thread pool ⑦. However, now, when the network call is issued, the control flow is detoured to a run-time policy that decides to put "network worker" thread to sleep for a period of time. During this period, the "background update pull" thread wakes up another "network worker" thread ⑧. Similar to the first "network worker" thread, the second instance is also put to sleep by the run-time. When the first "network worker" thread has been asleep for long enough (based on a deadline determined by the run-time) at ⑨, it and the second "network worker" thread are awakened. As a result, their two network requests are batched within a single cycle of the radio state machine. More detailed information about the run-time operation is given in Subsection 3.2.

2.3 Cross-application batching

Batching network requests is premised on the independence of the requests. In practice, there are two clear cases that network requests cannot be batched: (1) the requests are sent sequentially by the same thread; and (2) the second request depends on a result from the first. These cases limit the opportunities for batching within a single application; batching requests across applications provide a larger scope for finding opportunities to reduce tail time energy. However, cross-application batching requires that we be able to readily apply a policy to multiple applications developed by different vendors.

3 DelayDroid

DelayDroid has the goal of allowing us to readily apply batching policies, including those that operate across applications, to applications created by different vendors, that may be available only as binaries. OS changes and SDK updates are disallowed. If this goal is achieved, existing, unmodified applications can be combined with different policies for evaluation on everyday phones, and successful policies can be easily released for broad use.

We now describe the design and implementation of DelayDroid for Android apps, illustrating its compile-time and run-time operation, and its policy interface. Section 4 then gives our example cross-app policy. As we show in Section 5, DelayDroid can be applied directly to groups of applications in the Google App Store which can then be run together with the policy on an out-of-the-box phone.

3.1 Compile-time

The input to the compile-time processing is an Android apk file. Most existing Android apps are written in Java and compiled to dex format files[3)], which are included in the apk file. The dex-format contains bytecode which preserves the structure of the program, including classes, methods, and the relationships between classes. This gives the compile-time the necessary information to analyze and refactor the bytecode to add request batching mechanisms to the program. The DelayDroid compile-time then generates an optimized app into which a desired policy is embedded. This process is automatic–developers only need to provide the compile-time with apk files. No source code annotation or source code modification is required.

Figure 2 shows the compile-time process in more detail, and we follow its steps in our description below.

(1) Locating the network calls. In the Android app frameworks, developers send network requests using two kinds of APIs: (i) system APIs provided by Android SDK, such as APIs in java.net, java.nio

3) Dex format. https://source.Android.com/devices/tech/dalvi-k/dex-format.html.

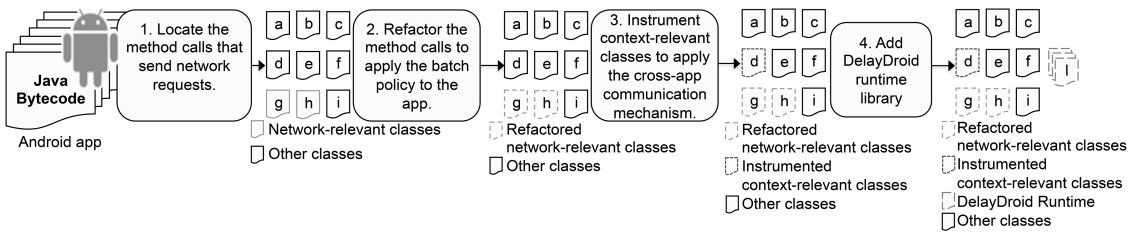

Figure 2 Compile-time process.

and org.apache.httpclient, and (ii) third-party APIs provided by third-party libraries, such as Volley and Retrofit. Because these third-party libraries are based on system APIs, and they are packed into the app as normal classes when developers build the app, we only need to focus on the system APIs.

To locate all the network calls, methods that send requests using the system APIs, the process essentially matches the method calls in the bytecode with the methods in the Android Network Library. We include a configuration file that notes which specific system APIs we intend to handle. Two forms of network calls are handled. The first are short-lived network calls (e.g., the APIs provided by Apache HttpClient), in which the server responds to a client request only once. The second form is the long-lived network call (e.g., the APIs provided by Socket), in which the client sets up a connection to the server, and uses the connection to send and receive multiple requests and responses. These two forms of network calls are located in the same way, but refactored differently, as we describe next.

(2a) Refactoring short-lived network calls. DelayDroid compile-time uses byte-code refactoring to detours the control flow of short-lived network calls by replacing the original network call with a call to a stub, NetworkStub, which is a class provided by the DelayDroid run-time. Figure 3(a) shows an example. Please note that the actual refactoring is done in Java bytecode—we show the decompiled Java code for convenience and clarity. In Figure 3(a), the line starting with "−" is the original code, while the line starting with "+" is the refactored version that replaces the original. Here, DelayDroid replaces the "execute" method call of HttpClient with the "execute" method call of NetworkStub. The refactored version uses the same method name as the original one with a different descriptor. The "execute" method of NetworkStub has an additional argument to convey the original argument to HttpClient.

Figure 3(b) shows NetworkStub. The "c.execute(q)" line executes the original network request (here with HttpClient). Note, however, that it also includes other method invocations prior to and subsequent to this step. These are interactions with the policy, named here DelayController. The first method, namely the "sleep" method of DelayController, causes the current thread to be put to sleep for a duration selected by the policy. The second method, namely the "wakeUp" method of DelayController, allows the policy to wake up other threads it has put to sleep, including those in other apps.

(2b) Refactoring long-lived network calls. Long-lived network calls return stream objects. For example, the "getOutputStream" method in Socket returns a OutputStream object. When its "write" method is called, network packets are transmitted to the server. Here, the "write" method call is the network call we are particularly interested in. However, refactoring the "write" method call as we might for a short-lived network call is infeasible for the several reasons:

• Stream objects are widely used. Not all "write" method calls of OutputStream objects, for example, send network packets. Even using alias analysis, the result is still inaccurate: may-alias information can cause false positive, while must-alias can cause false negative. Indeed, due to polymorphism, whether a particular "write" involves the network is only known at run-time.

• Widely-used utility classes provided by system library, such as BufferedOutputStream, may also call the "write" method of stream objects. Since the implementations of these classes are not part of the apk, we cannot refactor them at compile-time.

To deal with these challenges, we refactor methods that return stream objects to return the original stream object wrapped in a special stream object included in the DelayDroid run-time. Only streams created at call sites where it is clear the stream originates from a network connection context are wrapped. Figure 4(a) shows an example of refactoring a long-lived network call. As before the line starting with

```
public class BackgroundWorker extends Thread {
    public void run(){
        …
       sendNetworkRequest();
    }
    private void sendNetworkRequest(){
        //…prepare argument in the request
       - HttpResponse r = httpClient.execute(arg0);
       + HttpResponse r = NetworkStub.execute(httpClient,arg0);
        //…
    }
}
```

(a)

```
public class NetworkStub {
    public static HttpResponse execute(HttpClient c, HttpUriRequest q){
        DelayController.sleep();
        HttpResponse r = c.execute(q);
        DelayController.wakeUp();
        return r;
    }
}
```

(b)

Figure 3 Refactoring a short-lived network call. (a) A refactoring example of short-lived network call; (b) "execute" method in "NetworkSub".

```
public class LongLivedWorker extends Thread {
    BufferedOutputStreambos;
    public void LongLivedWorker(SocketAddressaddr){
        //…
        bos = new BufferedOutputStream(
                - socket.getOutputStream()
                + NetworkStub.getOutputStream(socket)
              );
    }
    public void sendData(String data){
        bos.write(data);
    }
}
```

(a)

```
public class NetworkStub {
    public static getOutputStream(Socket s){
        DelayController.sleep();
        OutputStream out = s.getOutputStream();
        out = new DelayedOutputStream(out);
        DelayController.wakeUp();
        return out;
    }
    //…
}
```

(b)

```
public class DelayedOutputStream extends OutputStream {
    private OutputStream out;
    public DelayedOutputStream(OutputStream out) {
        this.out = out;
    }
    public void write(int paramInt) throws IOException {
        DelayController.sleep();
        this.out.write(paramInt);
        DelayController.wakeUp();
    }
    //…
}
```

(c)

Figure 4 Refactoring a long-lived network call. (a) A refactoring example of long-lived network call. The returned value is a "delayed" output stream. (b) "getOutputStream" method in "NetworkStub". (c) DelayedOutputStream override all public methods of OutputStream.

"−" is replaced with the one starting with "+". Figure 4(b) shows how the implementation of the NetworkStub handles "getOutputStream", returning a DelayedOutputStream object which is a wrapper of the underlying OutputStream object. Figure 4(c) shows DelayedOutputStream. Here, the underlying "write" method is surrounded by calls to the "sleep" and "wakeUp" methods of DelayContoller, as with short-lived network calls.

(3) Instrumenting with cross-app communication. The cross-app communication mechanism is designed to support policies that batch network requests across apps. The mechanism consists of two operations: (i) sending messages to other apps to notify them that it is a proper time to send requests; and (ii) receiving such messages from other apps and acting in response. The challenge here is not with the mechanism, but with adding it to existing applications developed by different developers.

To implement these two operations, we leverage a publish/subscribe mechanism, namely Android BroadcastReceiver[4]. We define a new broadcast message called the Wakeup message and we register a corresponding receiver in each app to receive Wakeup messages. While this is simple, for security purposes, normal classes cannot communicate via such broadcast messages without a shared Context Object[5]. We introduce shared access to the Context Object through compile-time refactoring.

As described in the Android Developer Guide, the Context Object can be accessed by ContextWrapper Object, which has 31 subclasses including two kinds of subclasses: Activity and Service. Activity and Service are fundamental parts of Android application model. When the app starts, a class that extends from Activity or Service is instantiated, and the "onCreate" method of it is called. In this methods, the "this" keyword points to an instance of Activity or Service, which is a ContextWrapper Object.

The DelayDroid compile-time first analyzes the app statically and constructs the inheritance chain[6] for every class in the app. For a given class, its inheritance chain contains all its parent classes in the app and stretches back to a class in system library. By comparing all of the subclasses of ContextWrapper in the Android SDK with the inheritance chain of each class, DelayDroid can find all the classes that extend

4) Android BroadcastReceiver. http://developer.Android.com/reference/Android/content/BroadcastReceiver.html.
5) Android context object. http://developer.Android.com/reference/Android/content/Context.html.
6) Inheritance chain. https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html.

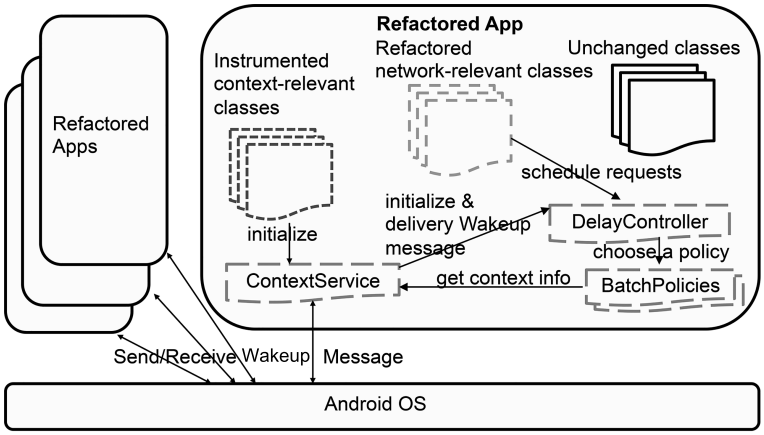

Figure 5 Run-time architecture.

from ContextWrapper directly or indirectly. Then, DelayDroid adds a method call at the beginning of the "onCreate" method in these classes. This call takes "this" as an input argument, and initializes the ContextService class within the DelayDroid run-time. ContextService is then responsible for sending and receiving the broadcast messages that allow for cross-app communication and thus policies, as described in detail in Subsection 3.2.

(4) Adding the run-time. In the final step, the DelayDroid run-time, as well as the refactored and instrumented classes from the original application are reintegrated into a new application. This apk file can then replace the original.

3.2 Run-time

Figure 5 presents the run-time architecture of DelayDroid and a group of refactored apps that are working with it. There are four kinds of classes in a refactored app: (1) refactored network-relevant classes (those with short- or long-lived network calls), (2) refactored context-relevant classes (those that are tracked in order to enable broadcast communication between DelayDroid-enabled apps; (3) DelayDroid run-time classes, which implement communication and pluggable policies; and (4) unchanged classes. The arrows between different classes denote the control flow of the system.

The DelayDroid run-time consists of ContextService, DelayController and BatchPolices. ContextService is in charge of sending and receiving the Wakeup messages. In addition, ContextService can provide the Context Object to a policy to allow it to access information in other contexts. When the app starts, context-relevant classes are instantiated. Then the instrumented code is executed to initialize ContextService. During its initialization, ContextService registers a Wakeup message receiver for the app. Finally ContextService initializes DelayController.

DelayController is in charge of batching the network requests of all DelayDroid-enabled apps on the system. During the initialization of DelayController, it chooses a batching policy according to a configuration file, and instantiates an instance of the policy. Operationally, before a network request is sent in any DelayDroid-enabled app, control flow is detoured to DelayController. DelayController then delays the request according to the policy and the other requests it knows about. Once a request is triggered and completed, the thread once again detours into DelayController which can then, according to policy, decide to wake up other requests as well, since at this point the radio is on.

3.3 Policy

The goal of batching network requests is to delay requests until the time that will minimize the energy wasted in the tail time. A good policy will minimize the occurrence of the tail, and when a tail does occur will maximize its communication value.

DelayDroid's policy interface, BatchPolicy consists of three methods: "start", "issue", and "finish". Before a request is sent, "start" is invoked, and it decides how long to delay the request before it is

Table 1 Classification of the 110 network calls currently handled by DelayDroid

Category	Description
HTTPWebkit	Methods that send Http request in package Android/webkit
HTTPApache	Methods that send Http request in package org/apache
HTTPJavaNet	Methods that send Http request in package java/net
TCPSocket	Methods that send TCP packets, for example, "send" method in class socket
UDPSocket	Methods that send UDP packets, for example "send" method in class DatagramSocket

actually sent, which it accomplishes by putting the invoking thread to sleep. When this time has passed, the request is actually sent. When a Wakeup message is received DelayController invokes the "issue" method of the policy, which can in turn decide to allow delayed requests to start at this point, prior to their normal timeout. The idea here is that if a request has been sent, the radio is on, so the policy may also want to let its sleeping threads make use of it. And after the request is done, the "finish" method is called to allow cleanup or to issue Wakeup messages.

BatchPolicy is an abstract Java interface. By implementing the interface, developers can write their own batching policies and deploy them to their apps easily. Policy implementations can also access ContextService and thus introspect on the DelayDroid-enabled applications as a whole. Section 4 will describe an example policy we have developed that seems quite effective.

3.4 Implementation details

In our implementation, we unpack the existing apk file, and refactor the dex bytecode using dex2jar[7]. A core aspect of the compile-time is the location of network calls. We handle 110 network calls from 16 classes in the Java SDK version 1.6 and Android SDK release 22. These network calls can be divided into five categories according to their protocol types and the Java packages they belong to, as summarized in Table 1. These calls can be expanded if necessary.

The DelayDroid compile-time consists of 2611 lines of Java source code, not including the ASM library. The DelayDroid run-time is written in 3652 lines of Java source code. This library is included in all DelayDroid-enabled apps, where it has a footprint of about 64 KB in the dex bytecode format.

4 Policy example

As part of our evaluation of DelayDroid, and as a contribution in its own right, we have developed an effective cross-app batching policy which we now describe. The policy is based on the observations of Huang et al. [2] who found that when the screen is off, there is a much higher chance that the user is not actively interacting with the device. We think it follows that at this time the network traffic is also most likely to be delay tolerant. Therefore, to minimize the visible effect on users, our policy only delays requests that are sent when the screen is off.

Our screen-status based policy, ScreenOffBatcher, operates as follows:

- If the screen is on, ScreenOffBatcher does not delay any requests.
- If the screen is off, requests are delayed by a default of 120 s. This batching window is intended to be a user-configurable value, similar to the screen timeout.
- When a request is sent by any app, ScreenOffBatcher assures that other delayed requests for it and other apps are also sent.

ScreenOffBatcher leverages ContextService in order to register a "screen status" receiver to get updates about the screen.

Figure 6 is the flow chart of the "start", "issue", and "finish" methods ofScreenOffBatcher. As is shown in Figure 6(a), when the "start" method of ScreenOffBatcher is called, it first checks the status of screen ①. If the screen is on, the network call is issued immediately ②. Otherwise ③, the run-time checks the timestamp of the last network call. If the last network call was finished within a threshold (10

7) dex2jar. https://code.google.com/p/dex2jar/.

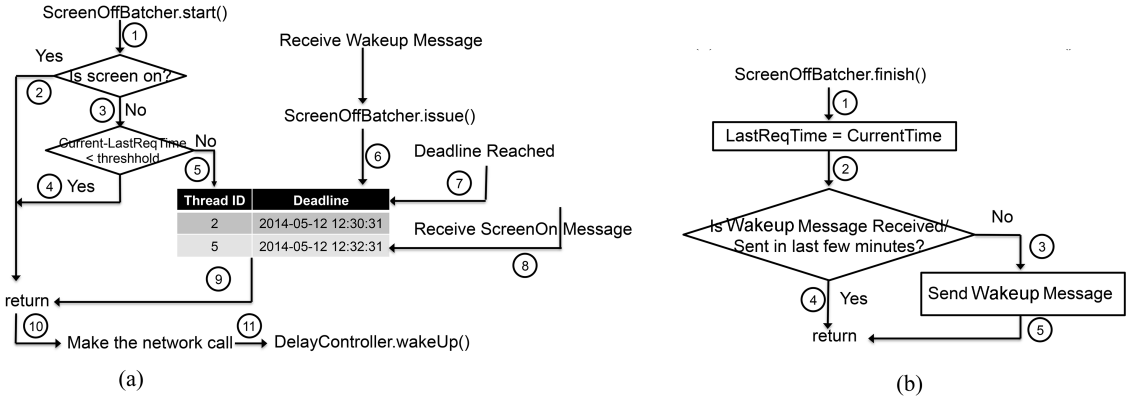

Figure 6 Policy example: ScreenOffBatcher. (a) Flow chart of ScreenOffBatcher.start(); (b) flow chart of ScreenOffBatcher.finish().

seconds by default), the current network request will be issued right away as well ④. If too much time has passed, the run-time puts the thread into a thread pool, and calculates the deadline for the thread to awaken, for example, 120 s from now ⑤. The sleeping threads in the pool will be awakened on three occasions:

- The wake-up message receiver in ContextService receives a broadcast message, which will call the "issue" method of ScreenOffBatcher ⑥;
- The earliest deadline in the thread pool is reached ⑦;
- The "screen-status" receiver receives a "screen on" broadcast ⑧.

After being awoken from the thread pool ⑨, the "sleep" method returns and control flow resumes at the network call ⑩. Finally, the control flow is detoured to DelayController.wakeUp ⑪, which invokes the "finish" method of the SceenOffBatcher policy.

Figure 6(b) shows the flow chart of the "finish" method. The ScreenOffBatcher policy updates the timestamp of the last network call ①. Then it checks the timestamp of the last Wakeup message to be sent or received ②. If this Wakeup message was received or sent within 120 s, it returns immediately ④. Otherwise, ScreenOffBatcher uses ContextService to send a Wakeup message ③ and returns ⑤. In this way, we can limit the time between Wakeup messages to at least 120 s, avoiding broadcast storms that might otherwise occur as we aggregate more DelayDroid-enabled applications.

While ScreenOffBatcher avoids impacting users by delaying requests only when the screen is off, it is important to note that this is a conservative policy, even in regard to users. For example, prior work [6] shows that users may tolerate slowdown even if they are interacting heavily with the phone while the screen is on. More aggressive policies could be readily employed within the DelayDroid framework.

5 Evaluation

Our evaluation of DelayDroid has two primary aspects. The first is the universality of the compile-time: can we indeed transform off-the-shelf apps into DelayDroid-enabled apps, and what is the overhead of doing so? The second aspect is the evaluation of the run-time, which we do together with our example cross-app policy: how well does it work for saving tail time energy?

5.1 Compile-time

We applied DelayDroid's compile-time toolchain to the fourteen off-the-shelf apps shown in Table 2. Please note that these are popular apps within the Google App Store that are available in binary (apk) form, and that our compile-time automatically transforms them into new apks that can replace the originals.

Table 2 Off-the-shelf apps from the google app store that we DelayDroid-enabled

App	Category	Number of users	App	Category	Number of users
Aws	Weather forecast	10000000–50000000	Mjweather	Weather forecast	1000000–5000000
Sohunews	News client	500000–1000000	Netease	News client	500000–1000000
Nytimes	News client	10000000–50000000	Taobao	E-commerce app	5000000–10000000
Pinterest	Social app	10000000–50000000	Renren	Social app	100000–500000
K9mail	Email client	5000000–10000000	Outlook	Email client	10000000–50000000
Pandora	Music app	100000000–500000000	Spotify	Music app	10000000–50000000
Mapquest	Map app	5000000–10000000	Translate	Google translate	100000000–500000000

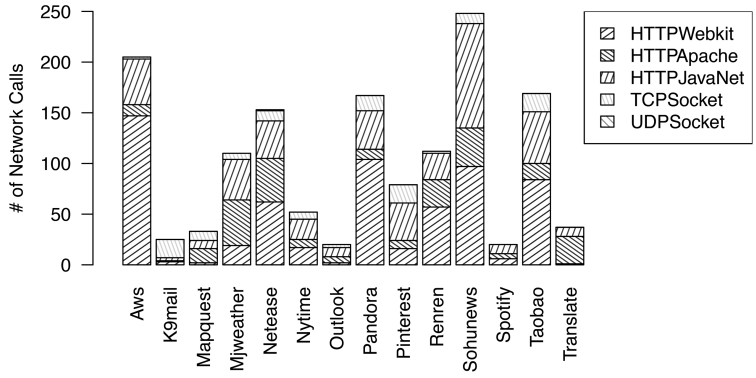

Figure 7 Network calls in different categories instrumented by DelayDroid.

The compile-time processing of DelayDroid is quite fast and produces very compact results. The process took from 28.8 s (Mapquest) to 92.2 s (Renren) on a typical development machine equipped with an Intel i5 3247U processor (1.8–2.6 GHz), 4 GB of memory, and a 128 GB SSD. This cost is paid only once—the resulting apk file can then be used in place of the original. The resulting apks were only marginally larger than the originals. Expansion was generally less than 1%, with the worst case being about 1.5% (K9mail and Translate). As one might expect, the percentage overhead in code size is inversely related to the original code size since typically the fixed size (64 KB) DelayDroid run-time library dominates the code expansion.

Figure 7 presents the result of the static analysis of each of our test apps, breaking down how many network calls in each of five categories are instrumented. HttpWebkit is widely used to fetch and render a web page as a part of an Android app. HttpApache and HttpJavaNet are popular for request/response communication because they are recommended in Android Developer Guide. Both are often used in third-party libraries, such as Volley and Retrofit, which further simplify networking in Android apps. TCPSocket provides a lower-level endpoint for reliable stream communication between client and server. Email clients such as K9mail use this to implement the IMAP and POP3 protocols. Pinterest and Pandora use TCP sockets indirectly. Pinterest uses Apache Thrift, while Pandora uses Java-WebSocket. Both of these third party libraries in turn build on SocketChannel within the standard library, which is a TCP socket. The unreliable datagram abstraction, UDPSocket, is rare—we saw it only in Netease.

The DelayDroid compile-time makes network calls slightly more expensive, as each call is treated indirectly, with wrapping calls to the policy. For a null policy, the overhead is essentially three extra calls per network call. However, since the cost of an underlying network operation itself is so much larger than that of a function call, the overhead of these additional calls is almost unmeasurable.

5.2 Run-time and example policy

The energy saved by the DelayDroid run-time depends on the apps, the policy and the network request pattern. When more apps run together, more network requests are sent, and thus there are more opportunities to delay and batch requests.

Table 3 App configurations for run-time experiments

App	Configuration
Aws	Add only one forecast location: Beijing
K9mail	Add an email account that created for the experiment; no new emails received every day
Sohunews	Default configuration

Table 4 Parameters of network energy model

Parameter	Value
Continuous downlink transfer power	1950.1 mW
Tail power	1060.0 mW
Tail duration	11576.0 ms

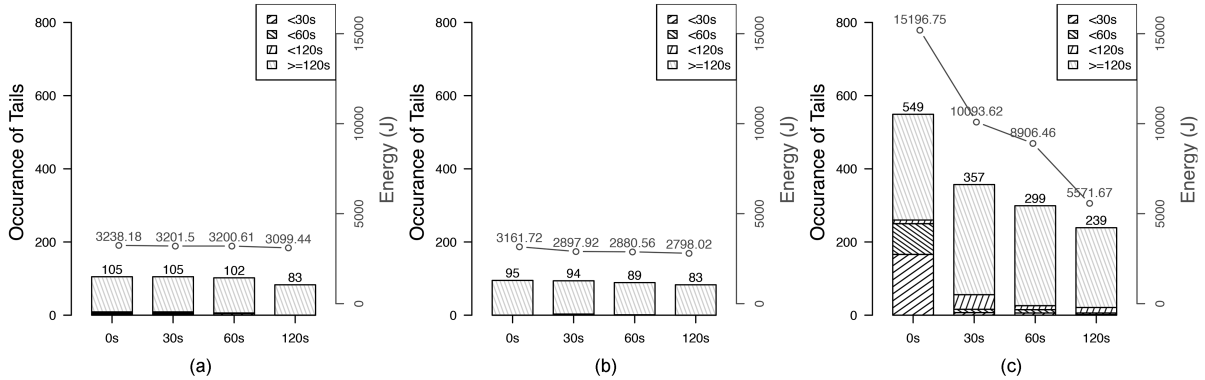

Figure 8 The reduction of radio energy consumption in each individual app as a result of reducing the occurrence of tails. (a) Aws; (b) K9mail; (c) Sohunews.

As the interaction between user satisfaction and delay/slowdown is outside of the scope of this paper, we selected a set of apps from Table 2 which can be reasonably be said to avoid this question. These apps, Aws, K9mail, and Sohunews, do not feature fine-grain interaction between the user and the network. Their configurations for our experiments are shown in Table 3. In our evaluation, we use the policy of Section 4, both operating on the individual apps, and on the apps operating together.

We run on Samsung Galaxy S3 mobile phones where network connectivity is 3G provided by China Unicom. The phones run CynaogenMod[8)], which is a variant of Android 4.2.2 that allows us more careful control over what processes are running on the phone. We use a fresh install plus the three DelayDroid-enabled apps. A Monsoon meter[9)] is used to measure the current flowing into the phone at a 5000 Hz rate. We are interested not only in the power/energy characteristics of the system as a whole, but also for the radio specifically. As we cannot measure the latter directly, we derive it from the system measurements using Huang et al.'s model [7], which is particularly suitable for our purposes as it focuses on screen-off scenarios, just as our example policy. Table 4 shows the parameters we use for the model.

5.2.1 *Radio energy savings for individual apps*

We measured the apps individually using batching window configurations of 0 s (disabled batching), 30 s, 60 s, and 120 s. In each case, the phone was kept screen-off, and measured for a period of 24 h.

Figure 8 presents the results. Each graph shows an individual app. The bars show the occurrence of tails (left vertical axis), while the line shows the radio energy (right vertical axis), both as a function of the batching window (horizontal axis). The bars are stacked, showing the rough distribution of tail lengths into four bins ($<$ 30 s, 30–60 s, 60–120 s, and $\geqslant$ 120 s). Shorter tails are evidence of greater energy waste. These are not common in Aws or K9mail, but much more so in Sohunews. As a result, our policy improves Sohunews considerably more than the others (although all do improve). With a batching

8) CyanogenMod. http://www.cyanogenmod.org/.
9) Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

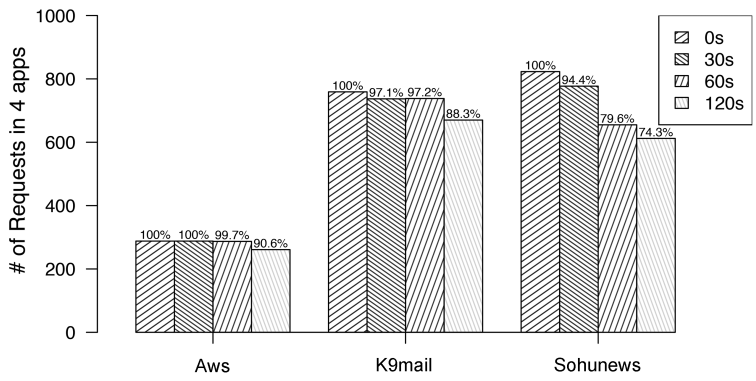

Figure 9 Reduction in the number of requests as a function of increasing batching window size.

window of 120 s, the radio energy consumption of Sohunews is reduced by 63% while the occurrence of the shorter tails is reduced by about 56%.

The radio energy reduction is due to two effects. The first is the reduction in the number of requests over any given duration (the request rate). Figure 9 shows this effect for each of the apps. For apps like Aws and K9mail, although we do not usually batch requests, the delay we introduce in responding to a request also delays the subsequent request. As a result, the request rate is reduced, and with it the occurrence of tails. For Aws and K9mail, this reduction is about 10% with a 120 s window. For Sohunews, the reduction is almost 26%, accounting for roughly half of its overall reduction.

The second effect leading to radio energy savings is batching requests, which, when the apps are considered individually, tends only to be common with Sohunews. Here it accounts for the other half of its overall reduction in "short" tails. With a batching window of 120 s, the number of tails with an idle time < 120 s reduces by about 92%, while the number of tails with an idle time $\geqslant 120$ s reduces by about 25%.

As far as we are aware, previous work does not consider radio energy savings based on this combination of effects because it assumes network requests are independent. In the case of these applications, they have sequential dependencies. The rate reduction made possible by this dependency, combined with batching, explains why the radio energy reduction we see in Sohunews is higher than in previous work.

5.2.2 *Radio energy savings for multiple apps*

We now consider combining applications. Given the results of the previous section, a natural question to ask is whether we can improve the radio energy savings for the Aws and K9mail apps by combining them to collectively produce more opportunities for batching. We will also consider combining all three apps. We consider three policy configurations: setting 1 indicates the policy is disabled, setting 2 uses the 120 s batching window determined to work well in the previous section, and setting 3 combines this with inter-app batching via the broadcast mechanism. Other than these changes, our methodology is identical to that of the previous section.

Figure 10(a) shows the results for the combination of the Aws and K9mail apps. When run individually, these apps each have a 10% energy reduction when run under our policy with a 120 s window. Combining them (setting 2) results in a 15% energy reduction. When the cross-app policy is fully deployed (setting 3), the energy reduction jumps to about 42%. This jump is due to a further reduction in the the occurrence of tails, which is due to batching that can now happen across the apps. Figure 10(b) illustrates what is happening in a strip plot of the requests. With setting 2, requests in the same app become batched, while with setting 3 they are more aggressively batched. The latter point is somewhat difficult to see on the plot since the combination of requests at timescales less than 120 s is difficult to differentiate—essentially, the vertical strips are just darker for setting 3.

Figure 10(c) shows the results of combining all three of Aws, K9mail and Sohunews. As noted in the previous section, batching requests in Sohunews itself produces considerable radio energy savings. This remains true when the three apps are combined. Setting 2 achieves 35% energy savings, while setting 3

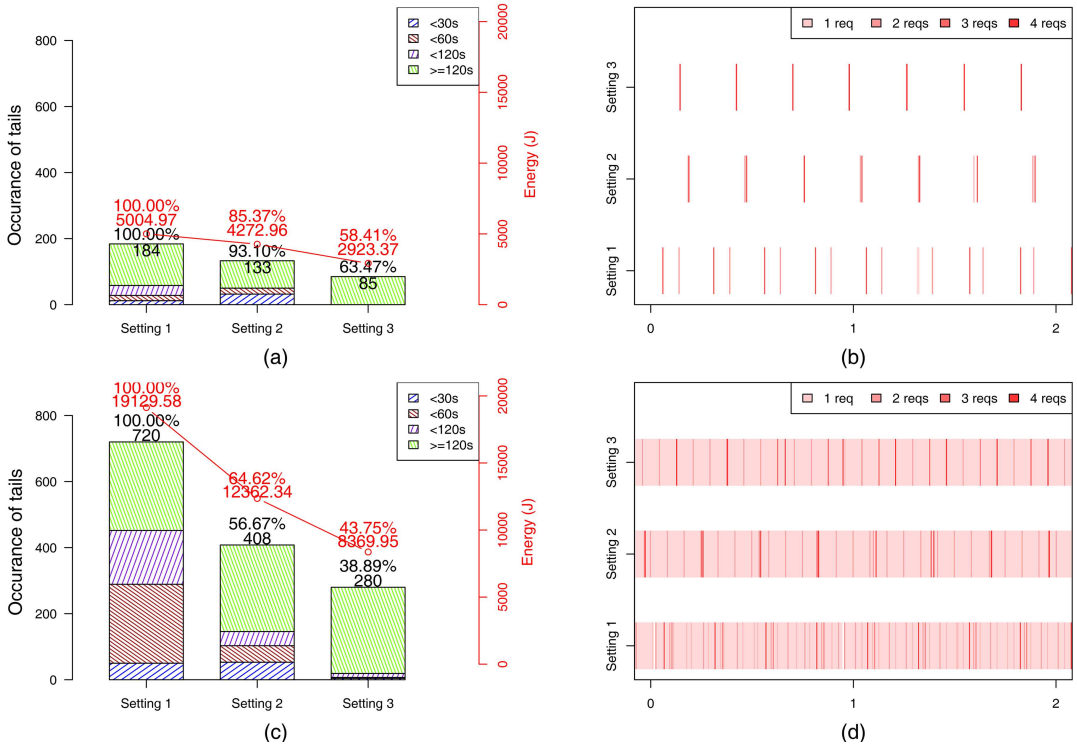

Figure 10 (Color online) Reduction of radio energy consumption for groups of apps using our policy. (a) Tail number & energy consumption of Aws & K9mail; (b) network requests of Aws & K9mail in 2 h; (c) tail number & energy consumption of Aws & K9mail & Sohunews; (d) network requests of Aws & K9mail & Sohunews in 2 h.

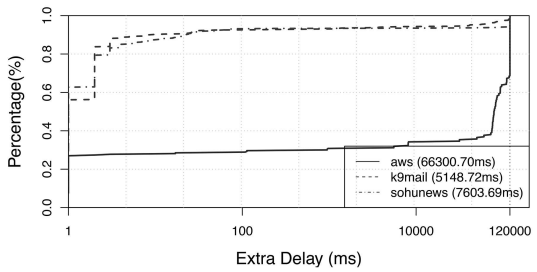

Figure 11 CDF of actual delays added by the policy for a batching window of 120 s.

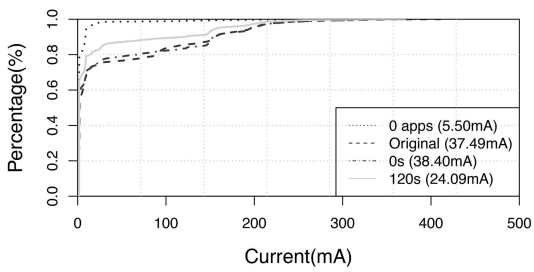

Figure 12 CDF of measured system current for different configurations.

achieves almost 56%. Recall that individually, Aws and K9mail achieved only about a 10% improvement. Figure 10(d) is a strip plot showing the dynamics of combining the applications, illustrating the increasing batching opportunities in settings 2 and 3.

Figure 11 gives a CDF of the actual delays inserted by DelayDroid and our policy during this experiment. Here the batching window is set to 120 s, the maximum we consider. The average delay Aws sees is 66 s, while K9mail sees 5 s, and Sohunews sees about 8 s. The differences are due to the different communication patterns of the apps and their structures (e.g., how many threads they use internally).

The results clearly demonstrate how by leveraging the cross-app communication and integration enabled by DelayDroid, our example policy can more effectively batch network requests from multiple apps and thus reduce the energy wasted in tail time. In practice, inserted delays are also quite small.

5.2.3 *Measurements of system energy*

The previous sections have presented results for the radio alone. We now present the direct system energy measurements of our example policy running across the three DelayDroid-enabled apps. Figure 12 shows CDFs of the current for four configurations:

• 0 apps is simply the phone without any applications running, and the screen off. The average current is 5.5 mA.

• Original is the phone running our three applications with the screen off. The average current is 37.5 mA.

• 0 s is the phone running DelayDroid-enabled versions of the three applications with the screen off and the policy disabled (0 s batching interval). The average current is 38.4 mA.

• 120 s is the phone running DelayDroid-enabled versions of the three applications with the screen off and the policy active (120 s batching interval and broadcast). The average current is 24.1 mA.

Our conclusion is that DelayDroid-enabling the group of three applications incurred an overhead of 1 mA ($< 3\%$), while allowing the creation and use of a policy that reduced current by 13.4 mA (36%).

6 Related work

There are numerous previous studies on reducing energy waste in tail time and these can be divided into three categories.

Policies and algorithms. A range of algorithms and policies have been proposed to reduce tail energy waste. Balasubramanian et al. [1] created TailEnder, a protocol that reduces tail energy waste by either delaying or aggressively prefetching network requests in applications that can benefit from such methods. However, applications must be classified by hand. In contrast, DelayDroid is able to delay network requests in arbitrary applications automatically. Work done by Huang et al. [2] has found that network activity when a phone's screen is off accounts for a large portion of the total network activity. Our example DelayDroid policy leverages this observation along with the fact that users should not be annoyed by delays introduced when the screen is off to decide how to batch requests.

The aforementioned prior studies are evaluated using traces from real-world apps, DelayDroid additionally has the potential to enhance the evaluation of such studies by making it possible to easily deploy the policies described in those studies and perform measurements directly.

Fast dormancy. Fast dormancy is another technology for optimizing the tail by allowing smartphones to immediately put the 3G radio in a low power state once all network communication has been completed. It extends the static RRC (radio resource control) state machine, and provides an interface to the developer to change the radio state dynamically[10]. Qian et al. [4] propose a simple interface, TOP, to allow different apps to leverage the fast dormancy feature. The app can explicitly invoke a tail removal API provided by TOP, which informs the cellular radio to quickly go into a low energy state. Athivarapu et al. [8] created 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. Similar to fast dormancy, several papers have proposed to dynamically choose idle timer values of the RRC state machine based on traffic characteristics instead of using static idle timer as is currently done [9, 10] .

These studies are complementary to DelayDroid. More energy savings could be achieved by combining them with DelayDroid.

Policy deployment and application development. Simplifying the deployment of policies for tail energy waste reduction (or fast dormancy) is an important challenge. One aspect is measurement. For example, ARO [11] is a tool that collects profiling data when an application is running (e.g., packet traces, user interaction, etc.) and provides developers with cross-layer information (TCP, radio states, etc.) on which to base policies. However, it is potentially costly for the developer to redesign his apps to directly embed such a policy. A tool like ARO could be used together with DelayDroid to allow wider deployment of policies that leverage detailed profiling data.

At the application layer, Xu et al. [12] have studied the power characteristics of email synchronization in smartphone apps. They developed five new techniques to optimize the energy consumption of the

10) Fast dormancy behavior. http://www.gsma.com/newsroom/wp-content/uploads/2013/08/TS18v1-0.pdf.

email client. While we feel that optimizing at the application layer can likely achieve the best result for individual apps, it leaves out opportunities to optimize across applications.

At the framework layer, Nikzad et al. [13] proposed APE, an annotation language and middleware service that eases the development of energy-efficient Android applications. Leverage APE, developers can apply the policy into their own application to batch network requests. However, it not only requires a lot of effort to identify all the network calls manually, but also misses the opportunity to batch requests in different apps.

Vergara et al. [14] created an OS-level packet scheduling mechanism using the Linux kernel Netfilter framework and implemented the Cluster Layer Burst Buffering algorithm [15]. Kernel-level deployment such as this allows a policy to be applied to all applications, but little application-level context is available in the kernel for the policy to use.

As for application development, there are various related studies. Zhang et al. [16] presented Dpartner, which refactors existing Android apps for on-demand computation offloading. Wu et al. [17] presented CoseDroid, an refactoring tool for Android app. By bytecode instrumenting, CoseDroid enables Android applications to virtually borrow computation or sensing resources from other devices. Ravindranath et al. [18] developed Procrastinator, an refactoring tool for Windows phone. Procrastinator can automatically decide when to fetch each network object that an app requests to minimize the data traffic wasted in prefetch. Dpartner, CoseDroid, Procrastinatorand DelayDroid are complementary. The difference is that DelayDroid uses similar technology to minimize the energy wasted in sending network requests.

DelayDroid focuses on enabling the design, implementation, and evaluation of framework layer policies within existing, unmodified, off-the-shelf applications. Unlike previous work, it provides a toolchain that automatically analyzes and instruments application binaries, and provides an interface that allows multiple-application policies for request batching to be incorporated. These policies can make use of the rich semantic information available at the framework layer to make their decisions.

7 Threats to validity

In DelayDroid, we attempt to automatically apply a policy within the apps through static analysis and bytecode transformations. Leveraging DelayDroid, developers can use the default batching policy or provide their own batching policy to save the energy wasted in tail time. However, it still meets the following limitations.

One limitation of DelayDroid is that DelayDroid does not provide any API to differentiate the delay-tolerant network requests from urgent network requests that are triggered by user. As a result, the default policy uses the screen status to distinguish the requests. A possible solution to this would be to utilize additional APK code information—in many cases network requests may be performed by events such as “onClick” or “onTouchEvent”, implying that they were launched by a user interaction and thus not delay-tolerant. By leveraging additional bytecode analysis DelayDroid could trace the origins of network requests, allowing for more fine-grained batching.

Another limitation of DelayDroid is that DelayDroid can only schedule the network requests that are sent in Java bytecode. In other words, DelayDroid cannot schedule the requests sent by native code such as C and C++. An extended version of DelayDroid could mitigate this limitation by also looking at any portions of apps that exist in LLVM[11] IR (Intermediate Representation), and refactor network calls made at that level at compile-time.

8 Conclusion

As the energy wasted in tail time becomes an important issue, many algorithms and policies are being proposed to address it. However, applying these policies to apps is challenging. We argue that asking an

11) LLVM. http://llvm.org/.

app developer to modify his code is not a winning proposition because it asks for altruism from the app developer to think hard about each network transfer and rewrite code around it.

In conclusion, we have implemented the first framework, DelayDroid, that can automatically analyze and refactor complex, existing Android apps, and expose the opportunity for batching network requests across apps to developers. DelayDroid has been demonstrated to operate successfully across a range of popular off-the-shelf apps available directly from the Google App Store. In addition, we have implemented a screen-status based policy as a default. An evaluation shows that using this default policy, DelayDroid can reduce the energy used by 3G by 5% to 63% in individual apps. When the cross-app communication mechanism is enabled, DelayDroid and the default policy can achieve energy savings as high as 41.6%. Moreover, in lab experiments we found that DelayDroid and the default policy can reduce the total measured system energy by 36%.

Acknowledgements This work was supported by National High Technology Research and Development Program of China (863) (Grant No. 2013AA01A208) and National Natural Science Foundation of China (Grant Nos. 61300002, 61421091, 61370020).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Balasubramanian N, Balasubramanian A, Venkataramani A. Energy consumption in mobile phones: a measurement study and implications for network applications. In: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, Chicago, 2009. 280–293
2 Huang J, Qian F, Mao Z M, et al. Screen-off traffic characterization and optimization in 3G/4G networks. In: Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement, Boston, 2012. 357–364
3 Qian F, Wang Z, Gao Y, et al. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In: Proceedings of the 21st World Wide Web Conference, Lyon, 2012. 51–60
4 Qian F, Wang Z, Gerber A, et al. TOP: tail optimization protocol for cellular radio resource allocation. In: Proceedings of the 18th Annual IEEE International Conference on Network Protocols, Kyoto, 2010. 285–294
5 Chuah M C, Luo W, Zhang X. Impacts of inactivity timer values on UMTS system capacity. In: Proceedings of IEEE Wireless Communications and Networking Conference Record, Orlando, 2002. 897–903
6 Swiech M, Dinda P A. Making javascript better by making it even slower. In: Proceedings of the 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, 2013. 70–79
7 Huang J, Qian F, Gerber A, et al. A close examination of performance and power characteristics of 4G LTE networks. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, 2012. 225–238
8 Athivarapu P K, Bhagwan R, Guha S, et al. Radiojockey: mining program execution to optimize cellular radio usage. In: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Istanbul, 2012. 101–112
9 Puustinen I, Nurminen J. The effect of unwanted internet traffic on cellular phone energy consumption. In: Proceedings of International Conference on New Technologies, Mobility and Security (NTMS), Paris, 2011. 1–5
10 Yeh J-H, Chen J-C, Lee C-C. Comparative analysis of energy-saving techniques in 3GPP and 3GPP2 systems. IEEE Trans Veh Tech, 2009, 58: 432–448
11 Qian F, Wang Z, Gerber A, et al. Profiling resource usage for mobile applications: a cross-layer approach. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, 2011. 321–334
12 Xu F, Liu Y, Moscibroda T, et al. Optimizing background email sync on smartphones. In: Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services, Taipei, 2013. 55–68
13 Nikzad N, Chipara O, Griswold W G. APE: an annotation language and middleware for energy-efficient mobile application development. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, 2014. 515–526
14 Vergara E J, Sanjuan J, Nadjm-Tehrani S. Kernel level energy-efficient 3G background traffic shaper for Android smartphones. In: Prcoeedings of the 9th International Wireless Communications and Mobile Computing Conference, Sardinia, 2013. 443–449
15 Vergara E J, Nadjm-Tehrani S. Energy-aware cross-layer burst buffering for wireless communication. In: Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet. New York: ACM, 2012. 24
16 Zhang Y, Huang G, Liu X, et al. Refactoring Android java code for on-demand computation offloading. In: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tucson, 2012. 233–248
17 Wu X, Xu C, Lu Z, et al. Cosedroid: Effective computation- and sensing-offloading for Android apps. In: Proceedings of the 39th IEEE Annual Computer Software and Applications Conference, Taichung, 2015. 2: 632–637
18 Ravindranath L, Agarwal S, Padhye J, et al. Procrastinator: pacing mobile apps' usage of the network. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, Bretton Woods, 2014. 232–244