Can Offloading Save Energy for Popular Apps?

Aki Saarinen, Matti Siekkinen, Yu Xiao,
Jukka K. Nurminen, Matti Kemppainen
Aalto University, School of Science, Finland
aki@akisaarinen.fi, {matti.siekkinen,
yu.xiao, jukka.k.nurminen}@aalto.fi,
matti.kemppainen@iki.fi

Pan Hui
Deutsche Telekom Labs, Berlin, Germany
pan.hui@telekom.de

ABSTRACT

Offloading tasks to cloud is one of the proposed solutions for extending battery life of mobile devices. Most prior research focuses on offloading computation, leaving communication-related tasks out of scope. However, most popular applications today involve intensive communication that consumes a significant part of the overall energy. Hence, we currently do not know how feasible it is to use offloading for saving energy in such apps. In this paper, we first show that it is possible to save energy by offloading communication-related tasks of the app to the cloud. We use an open source Twitter client, AndTweet, as a case study. However, using a set of popular open source applications, we also show that existing apps contain constraints that have to be released with code modifications before offloading can be profitable, and that the potential energy savings depend on many communication parameters. We therefore develop two tools: the first to identify the constraints and the other for fine-grained communication energy estimation. We exemplify the tools and explain how they could be used to help offloading parts of popular apps successfully.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Design studies]; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

Keywords

energy, offloading, smartphone, Android

1. INTRODUCTION

A seemingly straightforward way to conserve battery life of a mobile device is to migrate application execution partially to cloud. This technique is called *offloading* or sometimes *cyber foraging*. Several frameworks, including MAUI [4], Cuckoo [6], CloneCloud [3] and ThinkAir [7], have been developed to offload CPU intensive tasks. However, most popular apps involve also network communication, such as updates of social networks or remote data access, which consume a major part of the overall energy. Hence, it is justified to ask: Can offloading techniques help save energy, if applied to such apps?

Existing research provides very little answers to the above question since little efforts have been put on offloading tasks involving network usage. Indeed, almost all previous work focuses on alleviating the load by migrating heavy computational tasks to the cloud. To address this shortcoming, we investigate the feasibility of offloading network intensive communication, taking open source applications as examples. To this end, we use a specific framework, ThinkAir [7], for application partitioning and migration. It should be noted that our goal is not to develop our own offloading framework, but rather to study the feasibility of using existing solutions with typical apps.

ThinkAir, like many other existing frameworks supporting application partitioning on a method granularity, provides APIs for specifying which methods are allowed to be offloaded. This requires programmers with expert knowledge to manually select and annotate these methods. Using a case study, we observe that in practice there are several constraints, such as Java serialization issues, usage of callbacks, and access to native APIs that are tied to the device, which limit the ability of remotely executing a method. We find these constraints to be non-trivial and laborous to identify manually. We develop a tool to automate the identification of such constraints and provide analysis results from a set of apps. The results reveal that such constraints exist very frequently in typical apps.

We also note that the potential energy savings from method offloading are difficult to estimate, because communication energy consumption, a major factor in popular apps, depends on many factors. These include traffic patterns, network conditions, network type, and so on. We develop a tool to estimate communication energy consumption on a per-method level. It can be used for evaluating the energy-efficiency and performance of existing code at development stage, and can therefore guide programmers in improving application implementation for better energy-efficiency.

The following points summarize our main contributions.

1) We take the first look into the feasibility of offloading popular apps and analyze the constraints that may reduce the opportunities of energy-efficient offloading of communication related tasks.

2) We develop a toolkit to help analyze and increase the

MobiArch'12, August 22, 2012, Istanbul, Turkey.

potential in offloading such apps. [1]Specifically, we develop a tool for identifying offloading constraints using static source code analysis and another tool to profile the energy consumption of each method which relies on models that take traffic patterns and tail energy[2] into account.

2. USE CASE: ANDTWEET AND K-9

To gain insights into what it means to offload typical existing network-intensive programs, we study two popular open source Android applications. First, we take AndTweet, a popular open source Twitter client, and offload methods which send/receive network packets to/from the Twitter servers, using the ThinkAir framework. Second, we take a look into the feasibility of offloading network-intensive methods in one of the most popular open source Android email clients, K-9 Mail.

2.1 How can communication offloading save energy?

The concept that we call *communication offloading* in this paper, focuses on reducing communication cost on mobile devices. There are two ways to save energy when offloading parts of program responsible for communication related tasks. First way is to reduce the network traffic that need to be handled by the mobile device. This can be achieved, for instance, by offloading methods that handle communication with a server or other peers in P2P system. Such communication contains signaling traffic [1], part of which can be suppressed.

Second way is to optimize traffic patterns and improve overall latency and/or throughput. Packet interval patterns and throughput, for instance, have a significant impact on communication energy cost[11]. As a consequence, grouping packets to bursts is more energy efficient. Bursting benefits can be achieved, for example, when a group of methods, where each method fetches data, is offloaded. In the resulting sequence, only the end result will be downloaded to the mobile device in one burst, instead of downloading the data in several small bursts for each method.

2.2 Offloading setup

In all our experiments, we use the ThinkAir offloading system[7], which allows us to offload a selected set of methods to a more powerful remote machine which we call *surrogate*. We use a Google Nexus One with Android 2.3 as the mobile device and a virtual machine running the Android x86 port as the remote execution platform. ThinkAir requires the programmer to annotate which methods are "offloadable", i.e. can be migrated to be run in the surrogate machine. We choose the methods which are annotated as "offloadable" manually. While ThinkAir provides the possibility of dynamically deciding whether to offload an "offloadable" method call or not, depending on current conditions, we disable all dynamic decision making features and simplify the setup by defining that all "offloadable" methods are offloaded every time.

Upon starting an application, execution in the mobile application is given to the ThinkAir execution controller, which then sends an image of the application code to the surrogate. Whenever a method marked as "offloadable" is executed, ThinkAir execution controller transfers the execution to the surrogate.

In technical terms, when a method marked as "offloadable" is executed, the mobile device blocks execution, serializes the class instance of the called method and all of its arguments using Java's serialization APIs, and sends the serialized objects to the surrogate. The surrogate then deserializes these objects, invokes the specified method, and sends the returned value (or exception) back to the mobile device. When the mobile device receives the reply, it handles it and continues execution as if the return value would have been produced by running the same code locally.

2.3 Problems in AndTweet

In AndTweet we offload as many methods as possible, which communicate with the Twitter backend. In other words, we manually look for all places, which induce network traffic between the mobile device and Twitter servers, and mark all these methods as "offloadable" for the ThinkAir framework. Our goal is to reduce the amount of traffic and improve the traffic patterns to hence save energy.

Our process for identifying the "offloadable" methods is as follows: we first manually identify events in the user interface (UI) which trigger network requests. We then proceed to read the code and look for methods whose execution could be migrated to the surrogate, starting from the method which handles the UI event. We encounter several limitations why methods could not be straight-forwardly marked as "offloadable". Problems are related to 1) methods accessing local hardware, 2) methods whose migration to surrogate is not possible without modifications and 3) methods accessing state that is not correctly synchronized between the device and the surrogate, therefore causing unexpected behavior.

A remotely executed method cannot interact with the UI or other hardware resources, because these resources only exist at the client.However, AndTweet mixes application logic and handling of UI in many of its methods. This results in application logic, which might be offloadable on its own, being tied to the device, because UI handling renders the whole method non-offloadable. To overcome the restrictions of UI interactions, we attempt to offload the next level of methods in the call tree, i.e. methods which are directly invoked from the UI handling code. Here we find Twitter-specific abstractions, such as a timeline, which contains list of recent tweets, and friend lists. This set of methods, however, contains another category of issues.

In order to migrate the execution of a method, the offloading system must of course transfer the related state information to the surrogate. In case of ThinkAir, the encapsulating class of an "offloadable" method must be serializable using Java's serialization APIs. In addition, the method may not access any state outside the serialized context, because this state will not be synchronized between devices. AndTweet implements the Twitter timeline processing with non-serializable classes. It stores some of its internal state, e.g. cryptographical tokens for authentication, using an Android standard library class SharedPreferences, which is not serializable. Similarly, HttpClient, the de-facto class for using HTTP in an Android application, is used for communicating with Twitter servers, and it is not serializable.

[1]Our tools are available under an open source license at `https://github.com/akisaarinen/smartdiet`

[2]Tail energy refers to the energy that is wasted when radio stays on for a timer specified amount of time after transmission has ended.

Measurement	Wi-Fi (avg/stdev)		3G (avg/stdev)	
	Original	Offloaded	Original	Offloaded
Total energy (measured)	2.67/0.59 J	25% less/0.3 J	3.92/1.97 J	18% more/2.1 J
Execution time for refresh to complete (measured)	2.69/0.59 s	6% more/0.5 s	3.86/2.02 s	33% more/2.1 s
Total traffic size (measured)	7.7/0.8 kB	17% less/0.5 kB	6.0/2.1 kB	31% less/1.8 kB
Energy used for network transmissions (estimated using model)	-	46% less/0.04 J	-	33% more/2.0 J

Table 1: AndTweet measurements for a single timeline refresh event. We are comparing the measurements for original AndTweet application and the version for which methods doing network communication with Twitter server were offloaded to surrogate. Original network energy estimate is not shown, because the model only gives relative numbers, not absolute energy quantities.

Methods using HttpClient are easily fixed by instantiating a new HttpClient in the surrogate. Existing instances no longer need to be migrated to either direction. SharedPreferences is more problematic, because it uses the local file system to save internal state, i.e. the preferences that need to be stored over application restarts. Instantiating a new SharedPreferences in the surrogate and blindly using it would result in different states between the client and the surrogate. Therefore, we need to implement a mechanism which synchronizes the remote SharedPreferences before and after a method is executed remotely. Otherwise the cryptographical tokens are out-of-sync between the device and the surrogate. The problems with state synchronization can be difficult to resolve because in the worst case no errors are reported. Instead, the method just does something unintended and the application, not to mention the developer, is unaware of it.

Detecting and fixing the aforementioned problems with AndTweet was difficult because classes and methods had a large number of dependencies. As the number of dependencies increase, so does the likelihood of a method depending on a trouble spot, which, in turn prevents the offloading.

Our experiences suggest that manually identifying the methods having offloading problems is non-trivial. It may include many cycles of trial-and-error and the only way to know whether all issues have been resolved is to test the application to see whether it works correctly or not. It is essential to have tools that can automate this procedure and guide the programmers into developing more offloadable code.

2.4 Energy consumption of offloaded AndTweet

After eventually offloading Twitter communication, we measure the energy consumption of non-modified and offloaded AndTweet, under two different network conditions. In first setup the surrogate is in the same Wi-Fi network as the phone. In the second scenario, the phone used 3G as the access network.

We use a Monsoon Power Monitor (www.msoon.com) to measure the energy consumption of the phone. We also collect the packet traces and use models to estimate the energy consumed by the network interfaces. We use the model in [11] to estimate Wi-Fi energy consumption. As for 3G, we use a deterministic power model described in [10]. In the measurements and the estimates, we exclude the one-time cost of transferring the application image. The applications could be, for example, pre-installed to the offloading infrastructure.

The results in Table 1 show energy consumption of a single Twitter event which checks and fetches new tweets, and then displays them. We observe that offloading saves one fourth of the total energy consumed in the Wi-Fi setup. As expected, the savings clearly come from having less network traffic. However, the execution takes slightly longer when offloaded, which increases the energy consumed by the display. The results are very different when switching to 3G. More energy is consumed with the offloaded version even if less data is transmitted and received. Because the RTT in 3G is an order of magnitude longer than with Wi-Fi, the remote invocation takes more time. Combined with long 3G inactivity timer values, this causes the network interface to be in active high-power state during the whole invocation. Execution time also has a big variance, because 3G latencies vary depending on network conditions and the radio connection state when starting the transmission.

The conclusion of this experiment is that finding spots where communication offloading can yield notable benefits is non-trivial. Both network conditions and traffic patterns play a major role in the profitability of offloading. The estimation of potential benefits is difficult using mere intuition.

2.5 Offloading K-9

In our attempt to offload the communication in K-9 Mail, a critical problem prevents us from achieving this goal using ThinkAir. K-9 application architecture is heavily based on asynchronous callbacks. The fetching of new email messages, for example, is initiated by calling a method with a callback as a parameter. The callback is called if new messages are available, and it will update the appropriate internal data structures as well as the UI. However, these kind of callback-based APIs are not offloadable using ThinkAir. Callbacks are not serializable and can not be invoked from the surrogate. To continue offloading K-9, we would need to re-write major parts of its internal application logic.

To the best of our knowledge, none of the other systems can handle this situation either, with the possible exception of CloneCloud. Usually the only way to interact back to the phone is to use the return value or exceptions of the called methods. Asynchronous APIs break this assumption and simply do not work. While this is something that could be added as a feature to the offloading framework, this problem adds to the list of limitations that restrict the usability of current implementations in real-world programs.

2.6 Lessons learned

We learned two essential lessons from the above described case study. First is that in practice there are many constraints that prohibit using method offloading techniques and these constraints are not intuitive to detect. Hence, we need at least semi-automated ways of detecting and, if possible, alleviating such constraints.

Second, it is possible to save energy by offloading parts of typical popular apps, which mainly consume energy by communicating, instead of executing computationally intensive tasks. However, the savings depend on several things: network conditions, traffic patters (before and after offloading), and the type of network interface used were shown to have an effect, but there are certainly also others. Therefore, we need a way to accurately estimate the energy consumption of the communication before and after offloading in order to understand which parts of the app could yield energy savings when offloaded. We investigate both of these issues in more detail in the following two sections and propose preliminary solutions.

3. DETECTING OFFLOADING CONSTRAINTS

In order to understand how severe the offloading constraint issue is among existing apps, we analyze a set of popular ones. However, before reviewing those results, we first describe how we perform the constraint detection in an automated fashion.

3.1 Tool for static source code analysis

In order to identify offloading constraints, we develop a tool for static analysis on the application source code. For each method in the application, it points out problems that can prevent offloading unless the code is modified. We currently focus on heuristics that identify problems associated with our offloading setup, which is using the Android platform and Java Serialization API to implement the remote execution of methods. Nevertheless, similar heuristics can be crafted to other remote execution mechanisms, including the Android Parcelable mechanism used in Cuckoo [6], .NET serialization used in MAUI[4], or even the virtual machine based thread migration used in CloneCloud [3], because they all set some restrictions on what kind of methods can be offloaded.

Hardware constraints: The first set of constrained methods are those that require access to the hardware of the local device. We currently identify method as having this constraint if it tries to show, for instance, notifications to the user, update anything on the screen, vibrate the phone, access the Bluetooth, wifi or usb subsystem, and so on. We have identified a total set of 20 constrained subsystems while going through Android system APIs. If a method accesses one of the constrained Android system APIs, it cannot be offloaded unless the code structure is changed.

Software constraints: Our second set of constrained methods are those that cannot be migrated to the surrogate at all due to migration mechanism requirements, or those that cause unexpected behavior when executed remotely due to inconsistent states between local and remote execution environments.

Migration limitations are specific to the mechanism used, which in our case is the Java serialization APIs. For a method to be directly migratable, its encapsulating class as well as arguments and return type must implement the Java serializable interface. We also identify the methods which could be modified to be serializable with minor changes by changing the dependencies, e.g. super and member classes, to implement the Serializable interface. We exclude the library code, because for instance the Android SDK code cannot usually be easily modified, even if the changes would be simple.

Statistic	Median	Min	Max
Number of methods	431	121	4411
Directly migratable	0.17%	0.00%	3.70%
Migratable with minor changes	15.7%	0.00%	46.8%
Hardware access constraints	14.2%	2.28%	41.3%
Potential unexpected behavior because of access to file system	10.7%	0.00%	30.3%

Table 2: Constraint statistics for 16 open source apps.

Regarding unexpected behavior, our tool finds all methods that access the local file system using either Android's SharedPreferences mechanism or Java's File class. ThinkAir does not synchronize the file system and files in the surrogate are thus not the same as those in the device. This will often cause unexpected behavior for the program accessing files. This principle of finding problematic API calls can be extended to find problems related to other non-synchronized resources as well.

A potential solution to the synchronization issues is a system which automatically synchronizes all relevant state. This is notably a hard problem on its own, but in the context of offloading we are also concerned about energy usage of the synchronization. Until efficient automatic solutions are presented, the developer must be assisted in overcoming the problems manually.

Based on our experiences with AndTweet, even a richer set of rules could be established to detect different types of remote execution issues. For example, if a class contains an instance of the non-serializable HttpClient class, as described in Section 2.3, our toolkit could suggest removing the member instance and replacing it with a new instance of HttpClient created on-the-fly every time it is needed.

3.2 Statistics from open source programs

We next analyze a set of existing apps with our tool. Our method allows us to study only open-source software. Unfortunately many popular applications on Android Market are closed-source. To find similar applications, we went through numerous Android application listings, and selected programs that are non-trivial in size and include either communication or non-trivial computation. [3] Additionally we analyze also some platform applications that are shipped with Android operating system, like the web browser.

Results are shown in Table 2. Maximum of 3% of methods are directly migratable. If all of the identified and repairable trouble spots in the source code were fixed, it would enable the migration of up to 47% of methods. The results show that real-world existing applications are heavily constrained in terms of what can be offloaded. Developers need assistance to create offloadable applications in order to enable energy savings by means of offloading.

4. PROFILING COMMUNICATION ENERGY

In order to understand how much communication energy different parts of the app use, we develop a tool for accurate communication energy profiling. The tool implements a measuring and modeling setup for profiling and visualizing the energy consumption of a given application. Data is collected dynamically during the execution of the app and

[3]We used Android Market, popular source code sharing site Github, Wikipedia and numerous other sources.

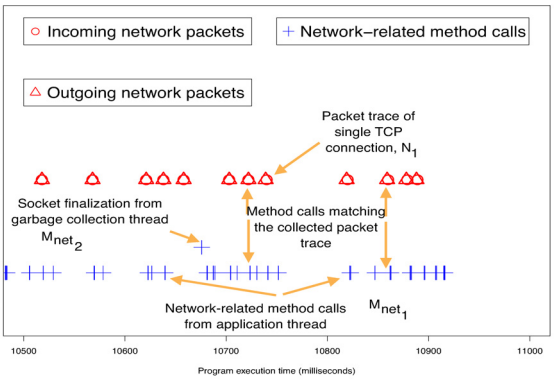

Figure 1: TCP packets and network-related method calls for test application fetching an HTML page.

analyzed afterwards. The tool collects two kinds of information: the traffic trace, and a trace of the program execution flow to later produce class and method level statistics. Packet trace is collected by a kernel module that captures and timestamps all traffic in the device using netfilter hooks (www.netfilter.org). To track the program execution, the tool uses execution tracking features in Android Debug Monitor Server (DDMS) which produces a trace of all classes and methods executed during the run. In addition, we implement minor modifications to the Dalvik virtual machine to annotate the traces with system-wide timestamps, which can be matched to those of our other measurements.

Our strategy for method-level communication energy accounting is as follows. We use statistical methods to associate each collected packet into one single method call in the app. Then, the energy consumed by the set of packets associated with a given method is estimated using power models same way as in 2.4. Finally, we recursively sum up the energy consumed by these methods as the energy consumed by their parent methods. In this way, we end up with a complete execution trace with communication energy consumed by each of the method. We detail and exemplify this procedure below.

To associate each packet in the collected packet trace to an individual method in the program execution trace, the tool first divides the execution trace into threads and the packet trace into separate flows (TCP connection or UDP flow). Furthermore, only network-related method calls are filtered for each thread. We now have two separate time series: network-related method calls of each thread and packet arrival events of each flow (see Figure 1 for illustration). These series are compared by computing cross-correlations in order to associate each flow to a particular thread which is generating that traffic. The idea is that each network-related method is associated with the corresponding packets. Finally, each packet of a flow is associated to the closest (in time) method call of the corresponding thread. This way, the tool generates a method trace of the program execution annotated with information about the methods that caused network traffic.

Figure 1 shows part of the traces of a simple Android test application that performs an HTTP GET request when a button is clicked. The thread executing the HTTP request correlates strongly with the packet trace. Another thread in the figure has a single network-related call. It is the garbage collector thread running finalization for a network-related object that is no longer used. Since it correlates weakly with the packet trace, no packets are associated with it.

Program execution in each thread can be viewed as a hierarchical call tree, where a method calls another method which calls another and so on. Our tool reconstructs this tree, carrying along the information of the detected network usage. It then aggregates the traffic of the nodes up in the tree, so that the root method, where the execution starts, gets associated with all packets that have been sent or received within each thread.

Figure 2 is an example graph, automatically produced by our tool, for a simple test case requesting an HTML document over HTTP. Traffic is cumulatively assigned to the MainActivity.onClick method and from there on, divided between various library functions that open the connection, send an HTTP request, receive the response and finally close the connection. At each step, we present the total number of calls made, the number of packets and the size of data generated by each call, alongside with model-based energy consumption estimates.

The energy usage estimate for each method is shown as a range between two values. In estimation one has to make assumptions about the interdependencies of methods within the program, which is why we show two numbers: a lower bound and an upper bound. The reason is that communication energy consumption is heavily dependent on traffic patterns meaning that only the exact number of bits transmitted does not determine the energy consumed but also their timing has a large impact (refer to [11], for instance). The lower bound corresponds to the energy consumed by the packets associated with the method in question, while the upper bound is computed so that it includes also the packets belonging to other methods that arrive between the first and the last packet of this method.

5. TOWARDS A TOOLKIT

The previous two sections describe our efforts so far to make it possible and feasible to offload parts of apps that fall into the typical apps category. Our overarching goal is to develop a complete toolkit for this purpose. We envision it to comprise three tools: energy profiler, constraint identifier, and structure analyzer. Above we described initial prototypes for the two first ones.

The idea is that the energy profiling tool finds and visualizes parts of applications that could yield energy savings when offloaded. The constraint identification tool identifies constraints in the source code, determines which methods can be offloaded as such and points out trouble spots in the code. The developer would first use the energy profiler to identify the candidate methods for offloading. Next, the constraint identifier would locate and possibly propose resolutions to trouble spots in the candidates. After modifications, the process can be repeated, eventually resulting in an application which is more suitable for offloading. The third tool could guide the developer in making larger structural changes, which would enable new portions of the application to be offloaded, in an energy-efficient fashion.

While we have focused here on the communication energy cost, computation can in some cases also be an important factor. We also plan to integrate CPU usage measurement and estimation to the same toolkit. Existing solutions can

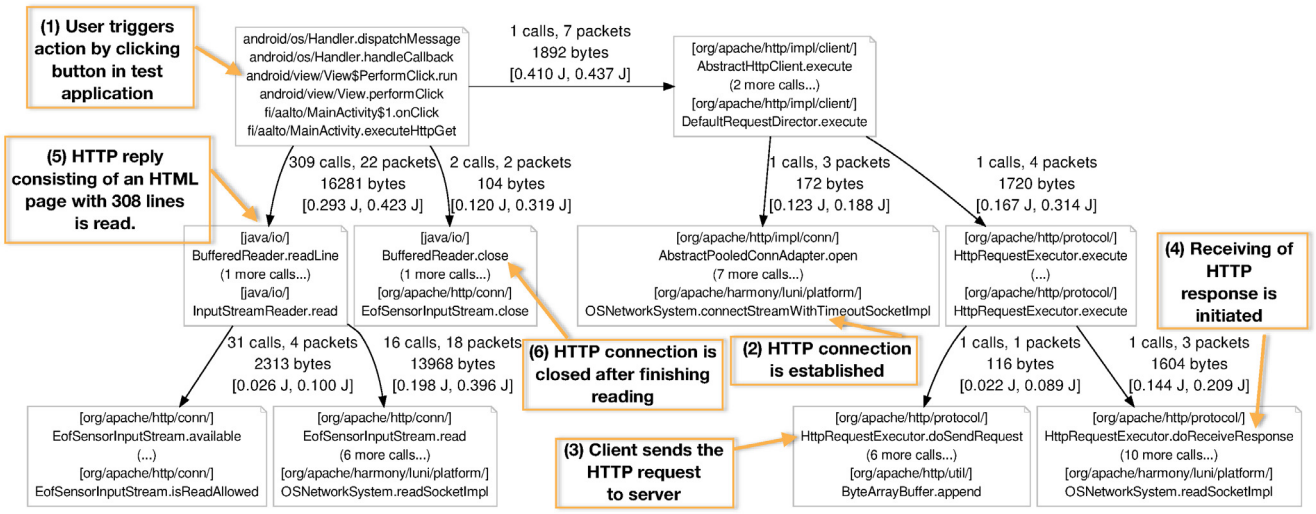

Figure 2: Network usage graph for test application, which fetches an HTML page. Numbers show how many times methods have been invoked during the whole procedure and how much energy it consumed.

be applied to estimate computation and display energy consumption[12] given just that time stamping and high enough sampling rates can be supported.

We encountered issues when analyzing the execution flow of complex programs which rely heavily on threading. We are currently looking into ways to statistically analyze the dependencies between execution of multiple threads. Alternatively, packet flow inside the program could be tracked, similar to that in TaintDroid [5].

6. RELATED WORK

Two main approaches have been suggested for *computation offloading* using application partitioning. MAUI[4], Cuckoo[6], Scavenger [8] and ThinkAir[7], for example, implement a framework on top of the existing runtime system. These systems only require access to the program source code and do not need any special support from the operating system. The second approach, used by CloneCloud[3], is to modify the underlying virtual machine or operating system. CloneCloud is a fully automated system and does not require having the source code of the program because it works directly on bytecode. Unfortunately, both types of systems have only been evaluated in the context of computationally heavy benchmarks, like N-queens puzzles [7] and virus scanning [3].

Similar with Eprof [9], our communication energy estimation tool takes tail energy into account. Furthermore, consider the traffic patterns, i.e. we take into account the fact that the power consumption in each active state of Wi-Fi interface increases with network data rate.

We share many concerns that Balan et al. have presented in [2]. Their goal was to enable rapid modification of applications for cyber foraging which is practically the same concept as offloading in a dynamic and opportunistic fashion. Their approach relies on the developer creating first a so called tactics file corresponding to the program being modified, after which the actual program code is modified, which is a fairly laborious process.

7. CONCLUSIONS

In this paper we studied the feasibility of using method offloading technique with popular apps to save energy. We used a set of open source apps to show that offloading using existing frameworks is often blocked by constraints, while also missing opportunities for saving energy. Fixing the problems manually and estimating potential energy savings is difficult. To this end, we propose initial solutions and a vision for a complete toolkit.

8. ACKNOWLEDGMENTS

This work was supported by the Academy of Finland, grant number 253860.

9. REFERENCES

[1] Smartphones and a 3G Network: Reducing the impact of smartphone-generated signaling traffic while increasing the battery life of the phone through the use of network optimization techniques. Technical report, Signals Research Group, LLC, May 2010.

[2] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, MobiSys '07, pages 272–285, New York, NY, USA, 2007. ACM.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[4] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an

information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[6] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Proceedings of MobiCASE*, Oct. 2010.

[7] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of IEEE INFOCOM 2012*, pages 945 –953, Mar. 2012.

[8] M. Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 217 –226, Apr. 2010.

[9] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[10] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, and P. Hui. Offloadable apps using smartdiet: Towards an analysis toolkit for mobile application developers. *CoRR*, abs/1111.3806, 2011.

[11] Y. Xiao, P. Savolainen, A. Karppanen, M. Siekkinen, and A. Ylä-Jääski. Practical power modeling of data transmission over 802.11g for wireless applications. In *e-Energy '10: Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 75–84, New York, NY, USA, 2010. ACM.

[12] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.