Reducing the Energy Consumption of Mobile Applications Behind the Scenes

Young-Woo Kwon and Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg, VA 24060
Email: {ywkwon,tilevich}@cs.vt.edu

***Abstract*—As energy efficiency has become a key consideration in the engineering of mobile applications, an increasing number of perfective maintenance tasks are concerned with optimizing energy consumption. However, optimizing a mobile application to reduce its energy consumption is non-trivial due to the highly volatile nature of mobile execution environments. Mobile applications commonly run on a variety of mobile devices over mobile networks with divergent characteristics. Therefore, no single, static energy consumption optimization is likely to yield across-the-board benefits, and may even turn to be detrimental in some scenarios. In this paper, we present a novel approach to perfective maintenance of mobile applications to reduce their energy consumption. The maintenance programmer declaratively specifies the suspected energy consumption hotspots in a mobile application. Based on this input, our approach then automatically transforms the application to enable it to offload parts of its functionality to the cloud. The offloading is highly adaptive, being driven by a runtime system that dynamically determines both the state-to-offload and its transfer mechanism based on the execution environment in place. In addition, the runtime system continuously improves its effectiveness due to a feedback-loop mechanism. Thus, our approach flexibly reduces the energy consumption of mobile applications behind the scenes. Applying our approach to third-party Android applications has shown that it can effectively reduce the overall amount of energy consumed by these applications, with the actual numbers ranging between 25% and 50%. These results indicate that our approach represents a promising direction in developing pragmatic and systematic tools for the perfective maintenance of mobile applications.**

I. Introduction

As mobile applications deliver increasingly complex functionality, mobile devices are rapidly overtaking the personal computer as a primary computing platform for an increasing number of users [6]. Because of the battery constraints of mobile devices, energy efficiency—fitting an energy budget and maximizing the utility of applications under given battery constraints—has become an important software design and maintenance consideration [14]. Traditionally concerned with performance and memory usage optimization, perfective maintenance now has to address the challenges of optimizing energy consumption, with existing perfective maintenance techniques being mostly inapplicable.

Although the field of optimizing software energy consumption is broad and diverse, existing solutions primarily focus on the hardware, operating system, and network layers. At the software layer, an energy optimization technique that has received a substantial attention from the research community is *cloud offloading* [4], [1], [17], [8]. This optimization partitions mobile applications, so that their energy intensive functionality is executed in the cloud, without draining the battery. Existing cloud offloading techniques determine the energy intensive functionality statically, and partition mobile applications accordingly. However, to achieve maximum benefit, cloud offloading must take into consideration the hardware capacities of the mobile device running the application as well as the characteristics of the mobile network connecting the mobile device to the cloud. In other words, the offloading decisions should be made dynamically at runtime and continuously adjusted in response to the fluctuations in the mobile execution environment. Lacking this level of flexibility, existing offloading schemes are inapplicable as a general energy optimization approach for perfective maintenance.

In this paper, we present a novel offloading approach that combines the advantages of the prior state of the art both in partitioning mobile applications and in dynamically adapting mobile execution targets in response to fluctuations in network conditions. Our approach is realized as the following two major technical solutions: (1) a multi-target offloading program transformation that automatically rewrites a centralized program into a distributed program, whose local/remote distribution is determined dynamically at runtime; (2) a runtime system that determines the required local/remote distribution of the resulting distributed program based on the current execution environment. Combining these two solutions can effectively reduce the amount of energy consumed by mobile applications without having to change their source code by hand, thus optimizing them behind the scenes.

From the maintenance process perspective, our approach works as follows. The maintenance programmer marks the methods suspected of being energy consumption hotspots. Then, a series of program analysis techniques validates the programmer's input and automatically rewrites the application into a distributed application, whose local and remote parts can be flexibly determined at runtime. The flexibility in determining the distribution patterns at runtime is enabled through an elaborate checkpointing mechanism. Depending on the runtime execution environment, different portions of a program's state can be checkpointed and transferred across the network as required by the offloading strategy in place. The offloading strategy is determined by the runtime system, whose

responsibilities include: (1) managing a connection between the client and the server, (2) continuously estimating the energy consumed by the mobile device, (3) calculating which offloading strategy should be followed, (4) synchronizing the checkpointed state transferred between the server and client, and (5) ensuring resilience in the presence of failure due to network disconnections.

In our experiments, we have applied our approach to optimize the energy consumption of third-party, real-world Android applications. All the subject applications not only reduced their energy consumption, but also maintained their original performance characteristics. Our adaptive, multi-targeted cloud offloading approach can effectively reduce the amount of consumed energy. Specifically, our benchmarks and case studies demonstrate that our approach can reduce the overall energy budget of a typical mobile application by an amount ranging between 25% and 50%.

Based on our results, the technical contributions of this paper are as follows:

1) **A multi-target offloading program transformation** that makes it possible to postpone until the runtime the decision about which parts of the application should be executed locally or remotely.
2) **An adaptive cloud offloading runtime system** that determines optimal offloading strategies for the partitioned applications.
3) **An empirical evaluation of multi-target offloading** that shows the technique effective in reducing the energy consumed by real-world, third-party mobile applications.

The rest of this paper is structured as follows. Section II defines the problem that our approach aims at solving and introduces the concepts and technologies used in this work. Sections III and IV describe and evaluate our approach, respectively. Section VI compares our approach to the existing state of the art. Section V presents perfective maintenance guidelines for effective cloud offloading, and Section VII presents concluding remarks.

II. Problem Definition and Technical Background

In this section, we provide a technical background both for the problem our approach is intended to solve and for the major technologies our approach uses.

A. Problem Definition

The work presented here is motivated by an insight we have recently derived from an experimental study of the impact of distributed programming mechanisms on energy efficiency [9]. After discovering that the network conditions in place can significantly affect how much energy is spent to transfer the same data, we have created guidelines for mobile application designers to transmit data in a fashion that consumes the least amount of energy for a given mobile network. Because transferring the same data over WiFi, 3G, or 4G networks consumes different amounts of energy, an optimal offloading mechanism should be adaptive, transferring varying amounts of state depending on the network conditions in place.

Figure 1 shows an Cloud Offloading Energy Consumption Graph (CO-ECG), a novel program analysis data structure that we introduced to model how much energy will be consumed under different offloading scenarios. The nodes of the CO-ECG represent program components, encapsulated units of functionality that can be offloaded to the cloud. Each node is labeled with an approximate amount of energy consumed by the CPU to execute the functionality of the component and its successor components in the graph. The edges represent the communication between the components, with the labels showing how much energy will be consumed by the mobile device to transmit the data between the connected components.

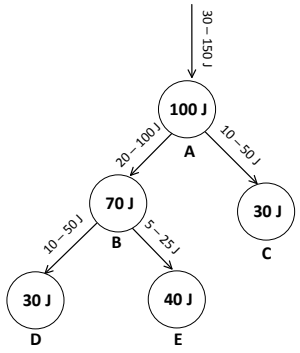


Fig. 1. An cloud offloading energy consumption graph.

In this particular CO-ECG, component `A` consumes approximately 100 joules during a typical execution, thus becoming a viable candidate to be offloaded to the cloud. Because component `A` calls components `C` and `B`, which in turn calls components `D` and `E`, its energy consumption is the sum of the energy consumed by all the successor components in the graph. We assume that the energy spent on executing the offloaded functionality in the cloud is free, as it does not exhaust any battery power of the mobile device. If component `A` is offloaded, then transmitting the necessary data to it across the network enabling it to execute remotely would consume between 30 and 150 joules depending on the type of the network available to transmit the data. In other words, under some network conditions, offloaded execution will end up using more energy than executing component `A` on the mobile device. Because the type of network available is only known at runtime, the offloading decisions must be dynamic to be able to optimize the amount of consumed energy under all runtime conditions.

In this paper, we present a solution that can solve the problem discussed above. We call our solution *adaptive, multi-target cloud offloading* (AMCO). A special program transformation creates a distributed client/server application, whose client and server functionalities are determined dynamically at runtime. As a specific example of using the CO-ECG above, when operating over a 3G network, components `C` and `D` can be offloaded, while when operating over a 4G network, only

component `E` can be offloaded. Finally, when operating over WiFi, components `A` or `B` can be offloaded.

More specifically, marking a method as an energy hotspot creates an offloading specification, in which various portions of the call graph rooted in the marked method can be offloaded as required by the runtime conditions. Because it takes more energy to transfer data across limited networks, an optimal offloading strategy needs to trade the energy potentially saved by moving the execution to the cloud with the energy consumed by moving the data (i.e., program state) to support the offloading. Our program transformation makes it possible to offload any subgraph of the CO-ECG, while our runtime system triggers the most beneficial (i.e., saving the most battery power) offloading for the runtime execution environment in place.

B. Technical Background

The technical concepts behind our approach include cloud offloading, energy consumption patterns, and program analysis. We describe these technologies in turn next.

1) Cloud Offloading: Cloud offloading is a mobile application optimization technique that makes it possible to execute the application's energy intensive functionality in the cloud, without draining the mobile device's battery. Cloud offloading is typically implemented as a program partitioning transformation that splits a mobile application into two parts: client running on a mobile device and server running in the cloud; all the communication between the parts is conducted via a middleware mechanism such as XML-RPC. Thus, cloud offloading is a special case of automated program partitioning—distributing a centralized program to run across the network using a compiler-based tool transform a centralized program [22] or migrating execution between different application images at the OS level [17], [2]. The promise of cloud offloading is demonstrated by the proliferation of competing approaches to this technique in the literature. CloneCloud [1] offloads execution at the thread level, while Cloudlet [17] offload at the VM level. MAUI [4] offloads through application partitioning at the method level. In our prior work on cloud offloading [8], we partition applications without destroying their ability to execute locally. All of these prior cloud offloading techniques share the goal of reducing the energy consumed by mobile devices. The approach presented in this paper adopts many of the techniques above to automatically transform mobile applications without any changes to their source code and to synchronize program states between partitions. However, our approach's goal is to improve on the efficiency of the prior cloud offloading technique by postponing the offloading decisions until the runtime, when a feedback-loop mechanism can determine which amount of offloading is optimal.

2) Energy Consumption Patterns in Mobile Applications: Network communication constitutes one of the largest sources of energy consumption in a mobile application [15], [9]. According to a recent study, network communication consumes between 10 and 50% of the total energy budget of a typical mobile application [13]. Specifically, in our prior research [9], we measured and analyzed how middleware can significantly affect a mobile application's energy consumption. Our experiments assessed the energy consumption of passing varying volumes of data over networks with different latency/bandwidth characteristics. Then, we isolated how mobile applications consume energy to infer their common energy consumption patterns. The experimental results and systematic analysis conducted through that research inspired us to initiate the work presented in this paper.

3) Program Analysis: Program analysis codifies a set of techniques to infer various facts about the source code to be leveraged for optimization and transformation. Class hierarchy analysis (CHA) constructs a call graph in object-oriented languages. Dataflow analysis determines how program variables are assigned to each other [18]. Side-effect free analysis [16] determines whether a method changes the program's heap. In this work, we use CHA to compute the functionality to offload and the program state to transfer for a given offloading. To select optimal offloading strategies, we combine dataflow and side-effect analyses. Based on the results, a bytecode enhancer then rewrites the application without changing its source code. We used Soot [23] to implement our program analysis and transformations.

III. Offloading Energy Intensive Functionality

In this section, we present adaptive, multi-target cloud offloading (AMCO). We start by giving an overview of the approach and then describe its major parts in turn. We conclude with discussing the approach's applicability and limitations.

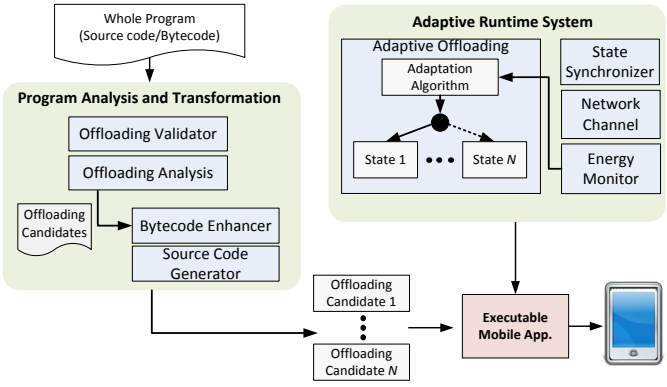


Fig. 2. Overall procedure for adaptive, multi-target cloud offloading.

A. Approach Overview

Figure 2 shows the workflow of AMCO. The approach consists of two major parts—a program analysis-guided partitioning transformation and an adaptive runtime system. The AMCO programming model is straightforward: the programmer marks the components suspected of being energy hotspots. In AMCO, components can be defined at any level of program granularity, with the smallest being individual methods and the largest a collection of packages. To mark

hotspot components, AMCO provides a special Java annotation @OffloadingCandidate; this information can also be specified through an XML configuration file. Based on this input, an analysis engine first checks whether the specified component can be offloaded as well as any of its subcomponents (i.e., successors in the call graph). The engine also calculates the program state, to be transferred between the remote and local partitions, that would need to be transferred to offload the execution of both the entire component or of any of its subcomponents. A bytecode enhancer then generates the checkpoints that save and restore the calculated state for the entire hotspot components as well as for each of its subcomponents. At runtime, an adaptive runtime system continuously monitors the energy consumed by each offloading candidate component, broken down for each of its constituent subcomponents. Based on the estimated energy consumption, the runtime offloads those subcomponents whose cloud-based execution would save the highest amount of energy for the network conditions in place. The runtime also synchronizes remote and local states in place (while preserving the aliasing in both the local and remote heaps). Yet another responsibility of the runtime system is fault tolerance—handling temporary network disconnection and volatility.

B. Program Analysis and Transformation

To create a program analysis heuristic that can calculate the program transformations enabling multi-target offloading, we have extended our prior heuristic for static offloading [8].

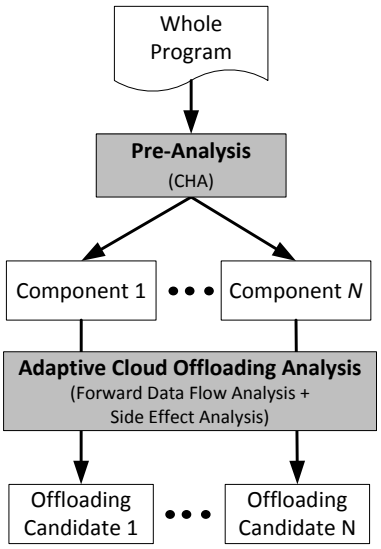


Fig. 3. Program analysis for adaptive, multi-target cloud offloading.

Figure 3 shows the analysis procedure that includes constructing a call graph by using class hierarchy analysis and identifying the state-to-be-transferred or synchronized by using forward dataflow and side-effect analyses. A pre-analysis step collects all the components and subcomponents marked with @OffloadingCandidate, and then the main analysis identifies the state that is needed to be transferred for each offloading scenario. One technical advantage of the main analysis is that it reduces the amount of state that has to be transferred across the network. The necessity to transfer large data volumes across the network can quickly negate the energy consumption benefits afforded by remote offloading. To avoid having to transmit the entire state, our approach leverages these forward dataflow and side-effect analyses to reduce the transferred state's size, thus rendering state transfer practical for energy optimizations. The algorithm examines assignment and invocation statements to determine whether the current state has changed during the cloud offloading.

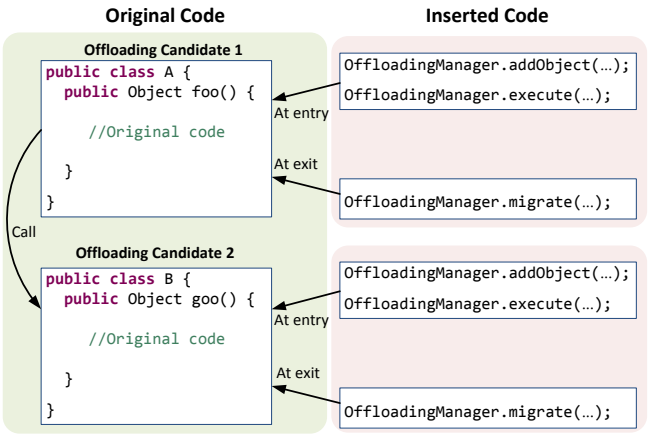


Fig. 4. An example of program transformation.

Once offloading candidates are identified, the bytecode enhancer transforms them to be able to run their hotspot components and subcomponents on the client and the server as needed to realize a given offloading scenario. Then, the adaptive runtime system monitors the runtime execution environment and determines an optimal offloading candidate. Figure 4 shows how the original code of a centralized mobile application is transformed. The bytecode enhancer inserts the code that can checkpoint and restore the necessary program state at the entry and exit of the potentially offloaded methods, respectively. These methods include offloading candidates and their successors in the CO-ECG.

C. Adaptive Runtime System

Figure 5 shows the design of the AMCO adaptive runtime system that comprises three main components: (1) cloud offloading prediction and steering, (2) energy consumption estimation, and (3) cloud offloading processing. By continuously monitoring the execution environment, the runtime system intelligently correlates the collected information to suggest optimal offloading strategies.

1) Cloud Offloading Prediction and Steering: Figure 6 shows the procedure for predicting and steering the mutli-target cloud offloading. The module predicts the future energy consumption by analyzing the collected runtime execution

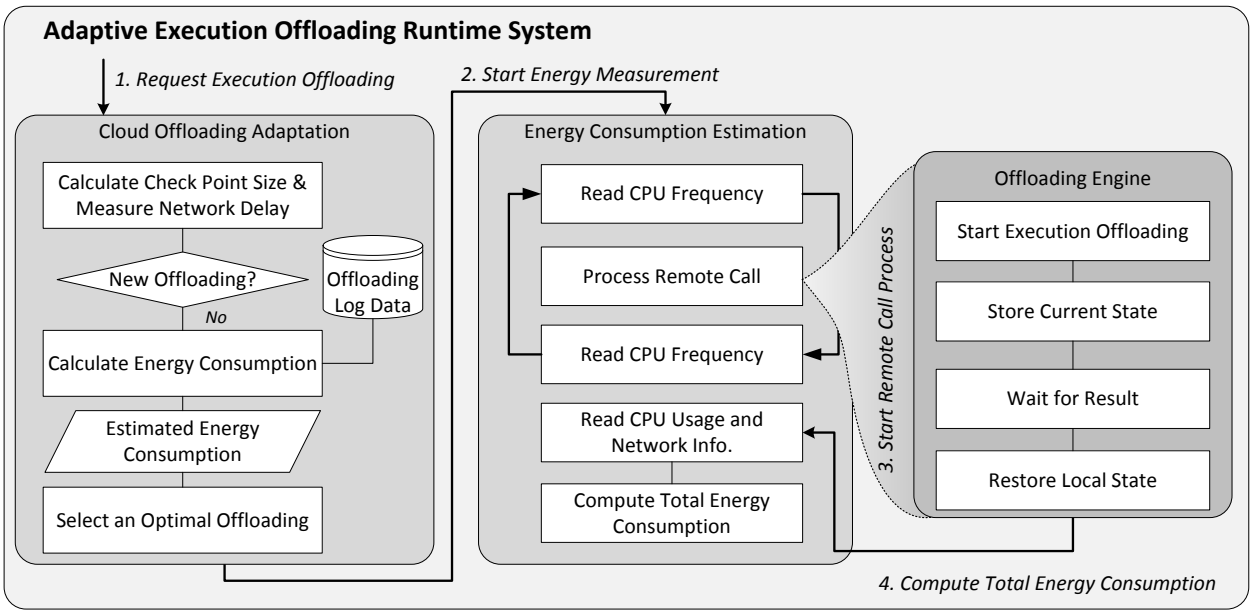


Fig. 5. Process details of adaptive execution offloading.

INPUT: A set of state, = $S_1, \ldots, S_N$
OUTPUT: Updated state

```
adaptiveOffloading() {
  delay ⟵ delay × α + currentDelay() × (1 − α);

  while(∀S) {
    estimation ⟵ computeEnergyConsumption(delay, S_n);

    if (estimation is the lowest) {
      if (estimation > local execution) {
        return ExecuteLocal();
      } else {
        newState ⟵ execute(cp);
        stateMigration(newState, state);
        updateHistory(state, delay, estimation);
        return state;
    } } }
}
```

Fig. 6. Adaptive, multi-target cloud offloading operation.

environment and then selects the offloading strategy that would yield the lowest future energy consumption. First, the future energy consumption is computed using delay, which is measured by sending a probe packet to the offload server. Then, the current delay is recomputed by giving a weight to the mostly recently obtained value (i.e., $delay = delay \times \alpha + delay \times (1 - \alpha)$), thereby avoiding a delay spike as well as partially reflecting the latest delay value. Finally, the module calculates the offloading strategy to suggest by picking the smallest predicted future energy consumption from all the available offloading candidates (i.e., subgraphs of the CO-ECG).

2) Energy Consumption Estimation: The energy consumption estimation module estimates the energy consumption before and after the offloading operation. To estimate the performed cloud offloading, we only compute the energy consumption by CPU and network communication as follows:

$$\boldsymbol{E} = \{P^{active}_{cpu\_freq} \times (T^{user\_time}_{cpu} + T^{sys\_time}_{cpu}) + (P^{active}_{net} \times T^{active}_{net}) + (P^{idle}_{net} \times T^{idle}_{net})\} \times V$$

where $P^{active}_{cpu_freq}$ is the power consumed by the CPU. Modern CPUs feature speed-step, a facility that allows the clock speed of a processor to be dynamically changed by the operating system, with different levels of power consumed at each clock speed. For example, Samsung Galaxy S III's AP provides five steps, ranging from 302.4 MHz to 1512 MHz, and the amount of power consumed at each speed ranges from 55mA to 577mA. $T^{user\_time}_{cpu}$ and $T^{sys\_time}_{cpu}$ are user and system times taken by the offloading operation, and they are obtained by consulting the statistics provided by the operating system (e.g., `\proc\[pid]\stat`). V is current voltage, which is obtained by using battery-related APIs (e.g., **class** BatteryStat in Android OS). P^{active}_{net} and P^{idle}_{net} are the amount of energy that the network processor consumes during the active and idle phases, respectively. For example, the active/idle energy consumption ratio for Samsung Galaxy S III is 96 mA/0.3 mA during WiFi communication, and 250 mA/3.4 mA during mobile communication (e.g., 4G). Finally, T^{active}_{net} and T^{idle}_{net} are active and idle time periods during the cloud offloading operation, measured at runtime. These device- and execution-specific values are used to compute the amount of energy consumed during each offloading optimization.

Another important responsibility of the runtime system is to

predict future energy consumption. To predict the energy that is likely to be consumed during an offloading optimization, it correlates the previously measured energy consumption and the current execution environment. Having obtained the current values of network delay, connectivity type, CPU frequency, and voltage, the future energy consumption is computed as follows:

$$\boldsymbol{E_{est}} = \{E_{cpu}^{avg} + (P_{net}^{active} \times T_{net}^{est\_active}) + (P_{net}^{idle} \times T_{net}^{est\_idle})\} \times V$$

where E_{est} is the predicted future energy consumption, E_{cpu}^{avg} is the average energy consumption of the given remote call, and $T_{net}^{est\_active}$ is the estimated communication time, which is computed by using the offloaded data size and delay, respectively. Finally, based on the estimated communication time and prior executions, the runtime system predicts the future energy consumption. The computed energy consumption is used for determining an optimal offloading strategy.

3) Cloud Offloading Engine: The cloud offloading engine manages a connection between the offload server and the mobile client, synchronizes the checkpointed state, and provides resilience in the presence of failure due to network volatility. The checkpointed state is synchronized by means of copy-restore, an advanced semantics introduced into remote method call middleware with the goal of passing as parameters liked data structures (e.g., linked lists, trees, and maps) [21]. Copy-restore copies all reachable state to the server and then overwrites the client's state with the server modified data in-place. To adapt to operating over cellular networks with limited bandwidth, we modified the original copy-restore implementation to use sparse arrays, which encode **null** values space efficiently. Our implementation uses **null** values to mark the portion of the transferred state that has not been mutated during the offloaded operations.

Figure 7 demonstrates how the runtime system synchronizes the checkpointed state. Graph (a) depicts the mobile device's state to be transferred to the server. The runtime system transfers only the nodes that the analysis identified as being used by the server (nodes 2, 3, 4, and 7). Graph (b) depicts the server's state before it is synchronized with the transferred state. Nodes 2, 4, and 5 are updated with new values; node 3 is reassigned to point to node 7. Graph (c) shows the synchronized server state. In this example, the offloaded server execution assigns a new instance, node 8 to node 3, modifies nodes 3 and 5, and assigns the **null** value to node 6, with Graph (d) depicting the results. The mutated state is then transferred to the client and synchronized with its state depicted in Graph (e). Specifically, node 6 is removed, node 3 is reassigned to point to node 8, and nodes 3 and 5 are overwritten with new values. Graph (f) shows the client state after the synchronization.

D. Discussion

In this section, we discuss advantages and limitations of our approach, adaptive, multi-target cloud offloading (AMCO).

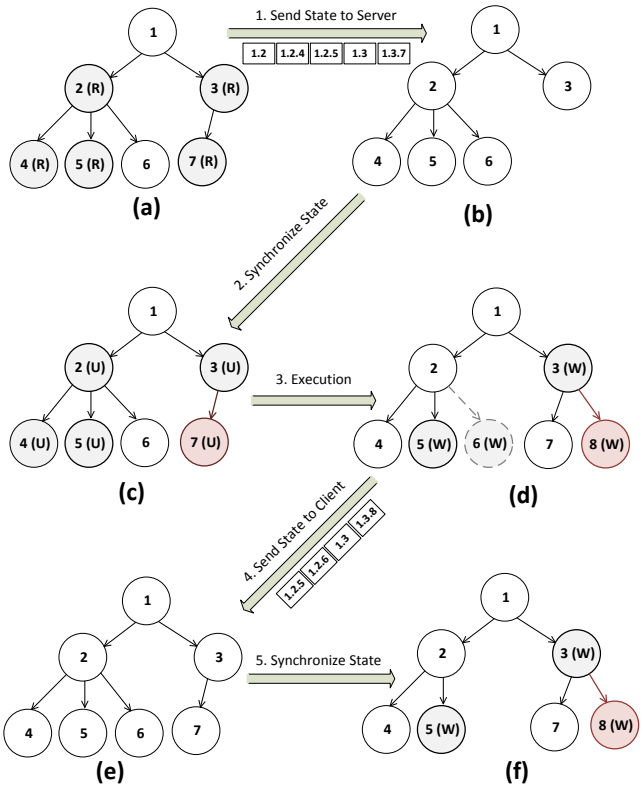


Fig. 7. Procedure to synchronize two different state.

1) Advantages: AMCO works with the standard, unmodified hardware/software stack; it employs bytecode engineering to transform programs and a lightweight runtime system to dynamically steer and adapt the offloading. AMCO makes it possible to keep the maintained version of the mobile application's source code intact, as only the bytecode version is transformed. AMCO requires a minimal programming effort, limited to marking methods as energy hotspots. AMCO makes offloading decisions at runtime by monitoring the execution environment, thus discovering optimal offloading strategies. Finally, the AMCO offloading transformations do not preclude the mobile application from executing locally in the case of the network becoming disconnected.

2) Limitations: Despite its advantages, AMCO cannot be applied to optimize the energy consumption of all mobile applications. The AMCO approach is automated rather than automatic; it relies on the programmer to identify the energy-intensive methods. Offloading at the method boundaries, AMCO relies on the subject applications following the well-accepted principles of quality software design (i.e., encapsulation, modularity, loose coupling), so that the offloaded methods encapsulate distinct functionality loosely coupled from the rest of the application. The AMCO offloading model also requires that the lifetime of all the threads in the offloaded methods coincide with the methods' boundaries. That is, the offloaded methods are free to employ multiple threads, but

all the threads are expected to terminate when the methods stop executing. As with all partitioning systems that rely on bytecode engineering, AMCO can only partition non-native (i.e., expressed exclusively in bytecode) state [20]. Finally, offloading can increase execution latencies, thus hurting the user experience for highly interactive applications.

IV. Evaluation

We evaluated our approach through a micro benchmark and a larger case study. The results show that our runtime system reports actionable environmental information without imposing unreasonable performance and energy overheads. Also, our overall approach can effectively reduce the amount of consumed energy for well-engineered applications, with the introduced program transformations and runtime execution never causing the enhanced applications to exceed their original levels of energy consumption.

A. Micro Benchmark

The purpose of this micro benchmark is to understand the overhead imposed by the runtime system, whose responsibilities include monitoring the relevant fluctuations in the environment, estimating potential energy savings due to the possible offloading steps, and synchronizing heaps during the offloading.

We have based our test suite on the benchmarks originally proposed by the JavaParty project [7], which is used widely in benchmarking middleware implementations. These benchmarks comprise remote invocations with varying parameter sizes and types. Similarly, our test suite assumes that a client needs to execute some server methods, each of which takes parameters that differ in their size and type. Because the executed server methods have empty bodies, one can reasonably attribute the energy consumed during their invocation to the underlying runtime system.

1) Experimental Setup: The experimental setup has comprised a Motorola Droid (600 MHz CPU, 256 MB RAM, 802.11g, 3G) and Samsung Galaxy III (1.5 GHz dual-core CPU, 2 GB RAM, 802.11n, 4G) as the mobile device and Dell PC (3.0 GHz quad-core CPU, 8 GB RAM) as the offloaded server. The mobile device has communicated with the server through WiFi, 3G network, and 4G network. For the WiFi connection, we have experimented with two emulated network conditions that have the following respective round trip time (RTT) and bandwidth characteristics: 2 ms and 50 Mbps, typical for a high-end mobile network and 50 ms and 1 Mbps, typical for a medium-end mobile network. For the mobile connection, we used a 3G network for the Motorola Droid and a 4G network for the Samsung Galaxy. Table I shows the energy profiles of these mobile devices. These device-specific values parameterize the runtime system.

To a large degree, it is the hardware specifications of a mobile device that determine its energy consumption profile. For example, the nano-level process technology is known to reduce the amount of energy consumed by modern CPUs. Similarly, the latest network and radio chipsets have improved their energy efficiency. Therefore, device-specific characteristics play a pivotal role in estimating the energy consumed by a given mobile device.

TABLE I
Manufacturer Provided Energy Profiles

	Samsung Galaxy III	Motorola Droid
CPU	1512.0 MHz: 577 mA	800.0 MHz: 280 mA
	1209.6 MHz: 408 mA	685.7 MHz: 236 mA
	907.2 MHz: 249 mA	571.4 MHz: 207 mA
	604.8 MHz: 148 mA	342.8 MHz: 165 mA
	302.4 MHz: 55 mA	228.5 MHz: 87 mA
	N/A	114.2 MHz: 66 mA
WiFi	96 mA	130 mA
	0.3 mA	4 mA
Mobile	250 mA	300 mA
	3.4 mA	3 mA

2) Energy Consumption Estimation: First, we evaluated how accurately the runtime system can predict how much energy will be consumed in a given time interval. Figure 8 compares the energy consumption predicted by the runtime system and that estimated by our model based on the actual resource usage. The average error is 10.6 % and standard deviation is 21.3 %. When considering only 90 % data removing outliers, the average error is 8.5 % and standard deviation is 6.8 %. These results indicate that the runtime system can predict the future energy consumption sufficiently accurately, with the discrepancies averaging 6-7%.

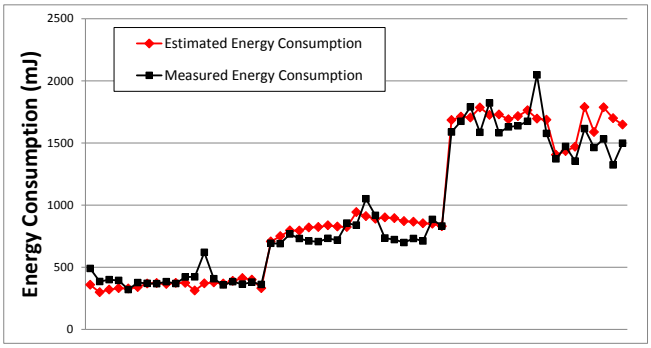
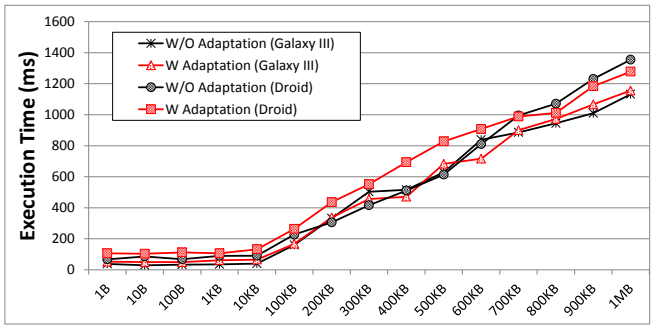
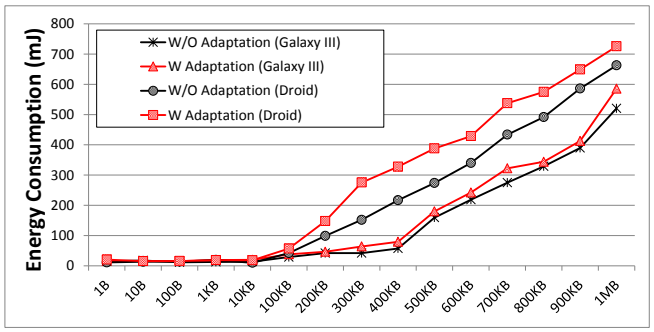


Fig. 8. Energy consumption estimation.

3) Overhead: In this benchmark, we compared the total execution time and energy consumption of the baseline versions of the benchmarks with that of in the presence of the AMCO runtime system. The first graph of Figure 9 shows the total execution time measured on two devices. As one can see, the performance overhead is quite insignificant. In particular, the overhead for both devices never exceeds 100 ms and remains constant for all the measured data transfer sizes. The second graph shows the energy consumed by each device. As expected, the high-end device (Samsung Galaxy) consumes less energy than the low-end device (Motorola Droid). Despite the low-end device having a larger overhead than the high-end device, the actual number was 100 mJ, a negligible value in comparison with the total energy typically consumed by a mobile application.



(a) Performance overhead.



(b) Energy consumption overhead.

Fig. 9. Performance and energy consumption overhead.

B. Case Study

To determine if our approach is applicable to real-world mobile applications, we experimented with open source projects as our experimental subjects. Pocket chess[1] is a mobile chess game, whose AI engine, contained in **class** SimpleEngine was marked as @OffloadCandidate. JJIL[2] is a face recognition application, whose recognition functionality, contained in **class** DetectHaarParam, was marked as @OffloadCandidate.

Figure 10 shows how the AMCO approach has reduced the amount of energy consumed by the subjects. For each subject, we present four graphs showing the amount of the energy consumed by typical, simple use cases. Specifically, for the chess application, we moved one randomly selected chess piece; for the face recognition application, we examined one image file for the presence of human faces. The use cases were performed under the following four optimization modes: (1) original centralized execution (baseline), (2) plain cloud offloading (offload the entire call tree rooted in the method marked with @OffloadCandidate, (3) same as (2) but with the transferred heap state minimized by means of program analysis, and (4) our approach, AMCO.

The optimized versions of the subject applications consumed less energy than their original and plain cloud offloading versions. A typical offloading strategy offloads to the cloud the heavy CPU processing required to calculate the next move, and transfers back only the new position for the piece to move. Because of this optimal architectural property of the chess application, its optimizations consume between 10% and 90% of the energy consumed by the original application. As the game proceeds, the optimized versions exhibit a constant rate of energy consumption, while the original version consumes an increasing amount of energy as the required AI processing intensifies. As expected, even the simplest energy optimization yields significant energy savings, without any tangible benefits afforded by using the AMCO adapative runtime system.

In the case of the face recognition application, all three optimization strategies showed different levels of effectiveness. First, the plain cloud offloading approach only can save energy under the most favorable network environment (Wi-Fi). Second, optimizing the amount of transferred state shows consistent improvement in the amount of consumed energy. Third, when the runtime is instructed not to monitor the environment, the amount of consumed energy in the offloaded version actually exceeds that in the original, centralized version. Lastly, the adaptive offloading capability of our AMCO approach seems to be pivotal to saving the amount of consumed energy consistently. In particular, our AMCO approach optimizes the application to consume between 10% and 80% less energy than the original local version, as well as between 25% and 50% less energy than a state-of-the-art static cloud offloading approach [8].

C. Threats to Validity

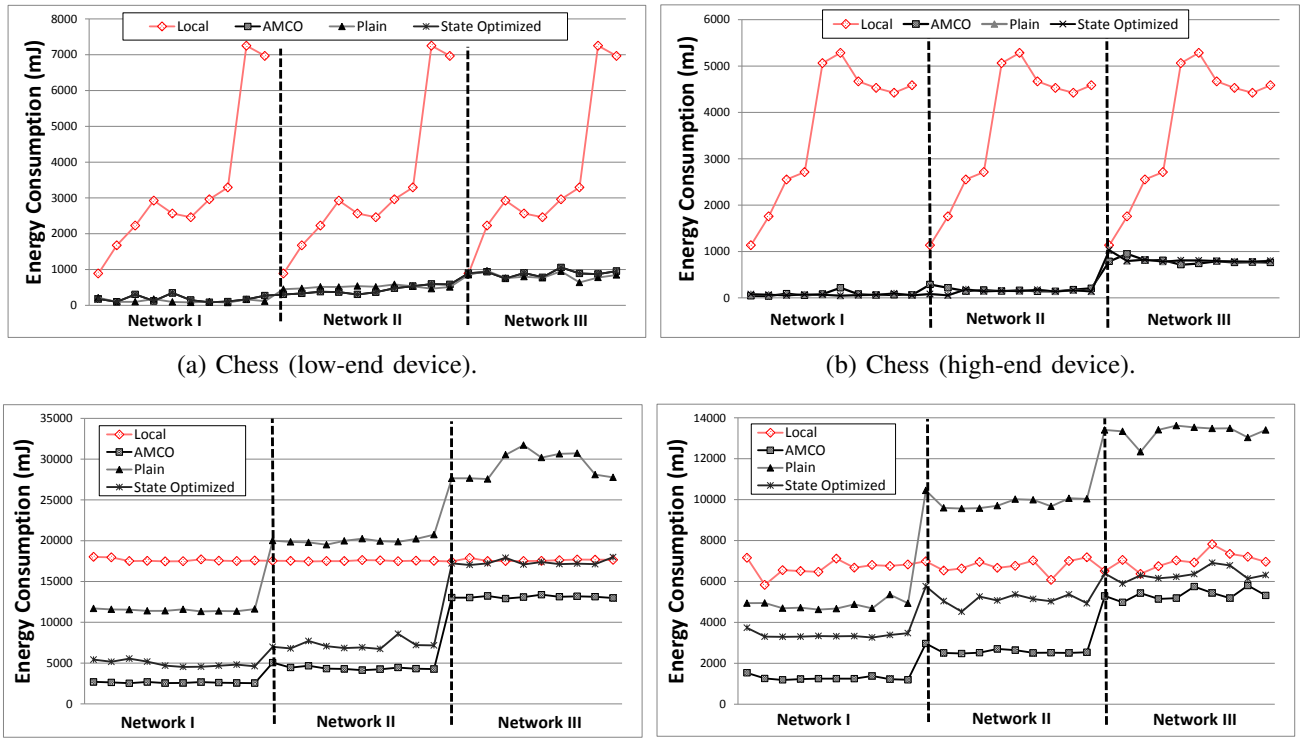
The experimental results above are subject to both internal and external validity threats. The internal validity is threatened by the manner in which the subject applications were run. Because the subject applications involve user interactions, their behavior and energy consumption depend on user actions (e.g., choosing a particular chess piece to move or taking a certain picture). These user interactions can heavily influence how much energy will be consumed.

The external validity is threatened by the mechanism employed by the AMCO runtime system to measure energy consumption. Rather than measuring the physical consumed energy directly, it estimates the energy consumption based on the actual resource usage information. As a result, the estimated energy consumption is likely to be less precise than those that would be measured through specialized hardware. In addition, the energy profiles that we used in the runtime system are provided by manufacturers, so that the accuracy of our measurements depends on the accuracy of the provided energy profiles.

V. Perfective Maintenance for Effective Cloud Offloading

When creating the AMCO approach, we have observed a positive correlation between the desirable software engineering properties (i.e., strong modularity, high cohesion, low coupling, etc.) of the application and its amenability for the cloud offloading optimization. Based on these observations, we next present three perfective maintenance guidelines that

[1]http://code.google.com/p/pocket-chess-for-android/
[2]http://code.google.com/p/jjil/



(a) Chess (low-end device).

(b) Chess (high-end device).

(c) Face recognition (low-end device).

(d) Face recognition (high-end device).

Fig. 10. Energy consumption of the subject applications.

programmer can follow to render their mobile applications better fit for effective cloud offloading optimizations.

a) Applying the Split Method Refactoring: A method is one of the earliest abstractions for separating concerns. A well-designed method should ideally contain a single coherent unit of functionality. In general, the larger the method, the less cohesive it is. As it turns out, large methods can be unwieldy as cloud offloading units, particularly in the case of fine-grained, adaptive offloading as in AMCO. In light of this observation, we recommend that the energy intensive methods marked with `@OffloadCandidate` be examined for their cohesion and then refactored if necessary. The Split Methods refactoring divides the functionality of a single method into separate methods calling each other. Smaller methods encapsulate conceptually coherent units of functionality. As a result, fine-grained, well-modularized mobile applications are not only desirable from the software comprehension perspective, but also are well amenable to perfective maintenance using cloud offloading.

b) Encapsulating Native State: Although native code usually renders the surrounding bytecode not modifiable [20], the programmer can guide the offloading tools by providing a bytecode API that accesses and synchronizes the portion of the state implemented in native code. Because native code may be impossible to analyze, it is the programmer's responsibility to ensure that the wrapping API correctly synchronizes the native state without introducing any harmful side effects.

c) Eliminating False State Sharing: In object-oriented languages, all the member fields of a class can be accessed by all of its methods. However, the purpose of member fields is to define the state, whose purpose remains constant throughout the lifetime of the class's objects. A common design flaw is to use the same member field across multiple class methods in different capacities. That is, rather than declaring the field locally in each method, several methods use the same member field, a condition that we call *false state sharing*. False state sharing is a more serious design flaw that it may seem on the surface, as the problems it causes are similar to that caused by global variables in procedural languages. With respect to cloud offloading, false state sharing complicates program analysis and state synchronization, thus limiting the amount of functionality that can be effectively offloaded. Therefore, we recommend that as part of perfective maintenance for energy optimization, programmers eliminate false state sharing through refactoring as much as possible.

VI. Related Work

Extending the battery life by reducing the energy consumed by mobile applications has been the focus of multiple complimentary research efforts such as more energy-efficient operating systems (e.g., energy-efficient CPU scheduling [26], disk power managements [24], and process migration [1], network protocols [25]), VM-level [17] and application-level offloading techniques [4].

Although the majority of these efforts has focused on one particular system layer (i.e., mainly the network), a technique called a cross-layer approach effectively controls energy consumption by leveraging the information provided by multiple system layers [5], [11]. Several programming abstractions enable effective adaptations that leverage multiple optimization strategies. The Odyssey platform [5] adapts data or computational quality to save energy consumption, so as not to exceed the available system resources. These energy-aware adaptations can identify possible trade-offs between energy consumption and application quality, choosing an energy management strategy based on the runtime conditions. DYNAMO [11] is another middleware platform that adapts energy optimization strategies across various system layers, including applications, middleware, OS, network, and hardware, to optimize both performance and energy.

While much research has focused on system-level solutions, programming-level approaches (e.g., algorithms [12], design patterns [10], software models [19]) have received little attention in the literature. A recent language-based approach to energy-aware programming is ET [3], a new object-oriented programming language that enables the programmer to write energy-aware code.

By contrast, the AMCO focus is on perfective maintenance, even though it adopts its techniques from both programing- and system-level approaches to energy optimization. The novelty of AMCO lies in combining analysis-driven program transformation and runtime adaptation.

VII. Conclusions

We have presented a novel perfective maintenance approach to reducing the energy consumption of mobile applications, adaptive, multi-target cloud offloading (AMCO). Our approach reduces the energy consumed by mobile applications without changing their source code by employing powerful program analysis and transformations as well as an adaptive runtime system that determines an optimal offloading strategy at runtime. We have evaluated our approach by reducing the energy consumed by micro benchmarks and third-party applications under different execution environments. These results indicate that our approach represents a promising direction in developing pragmatic and systematic tools for the perfective maintenance of mobile applications.

Acknowledgments

This research is supported by the National Science Foundation through the Grant CCF-1116565.

References

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the 6^{th} ACM European Conference on Computer Systems*, 2011.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2^{nd} conference on Symposium on Networked Systems Design & Implementation*, 2005.

[3] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2012.

[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 8^{th} International Conference on Mobile Systems, Applications, and Services*, 2010.

[5] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.

[6] Gartner, Inc. Predicts 2013: Mobility becomes a broad-based ingredient for change, Nov. 2012.

[7] B. Haumacher, T. Moschny, and M. Philippsen. The JavaParty project. www.ipd.uka.de/JavaParty, 2007.

[8] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32^{nd} International Conference on Distributed Computing Systems*, 2012.

[9] Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, 2013.

[10] Y. Liu. Energy-efficient synchronization through program patterns. In *Proceedings of the 1^{st} International Workshop on Green and Sustainable Software*, 2012.

[11] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *IEEE Journal on Selected Areas in Communications*, 25(4):722 –737, 2007.

[12] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenIT. In *Proceedings of the 1^{st} International Workshop on Green and Sustainable Software*, Zurich, Suisse, 2012.

[13] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7^{th} ACM European Conference on Computer Systems*, 2012.

[14] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.

[15] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9^{th} International Conference on Mobile Systems, Applications, and Services*, 2011.

[16] A. Rountev. Precise identification of side-effect-free methods in Java. In *Proceedings of the 20^{th} IEEE International Conference on Software Maintenance*, 2004.

[17] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[18] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[19] C. Thompson, H. Turner, J. White, and D. Schmidt. Analyzing mobile application software power consumption via model-driven engineering. In *Proceedings of the 1^{st} International Conference on Pervasive and Embedded Computing and Communication Systems*, 2011.

[20] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *Proceedings of the 5th International Conference on Generative programming and Component Engineering*, pages 89–94, 2006.

[21] E. Tilevich and Y. Smaragdakis. NRMI: Natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):174–187, 2008.

[22] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1–40, 2009.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*, 1999.

[24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. *ACM SIGOPS Operating Systems Review*, 36(SI):117–129, 2002.

[25] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21^{st} IEEE Computer and Communications*, volume 3. IEEE, 2002.

[26] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37(5):149–163, 2003.