Configurable and Adaptive Middleware for Energy-Efficient Distributed Mobile Computing

Young-Woo Kwon[1] and Eli Tilevich[2]

[1]Department of Computer Science, Utah State University
young.kwon@usu.edu
[2]Department of Computer Science, Virginia Tech
tilevich@cs.vt.edu

Abstract. The energy demands of modern mobile devices are outstripping their battery lives. As a result, energy efficiency—fitting an energy budget and maximizing the utility of applications under given battery constraints—has become an important system design consideration. Because network communication incurs high energy costs in mobile applications, middleware presents a promising target for energy optimizations. Unfortunately, mainstream middleware mechanisms are oblivious to the highly volatile nature of mobile networks, operating over which energy efficiently requires aligning the middleware communication patterns with the network conditions in place. In this paper, we present a novel middleware architecture that optimizes energy consumption by adapting various facets of middleware functionality (e.g., data communication, encoding, and compression) dynamically in response to fluctuations in network conditions. By means of a simple configuration file, the programmer can specify the policies to follow for various parts of the communication functionality and how policies should be triggered by changes in network conditions. As compared to mainstream middleware mechanisms, our reference implementation improves the energy efficiency of mobile applications. Specifically, our benchmarks and case studies demonstrate that the new middleware architecture can reduce the energy budget of a typical third-party mobile application by as much as 30%.

Key words: energy-efficiency, mobile computing, energy models, middleware, dynamic adaptation

1 Introduction

Energy efficiency is rapidly becoming a key software design consideration [13], as mobile devices are steadily replacing desktop computers as the dominant computing platform. The increasingly feature-rich nature of mobile applications renders battery capacities a key limiting factor in the design of mobile applications [15]. To reduce the energy consumed by modern mobile applications, system designers must consider all the constituent parts of a distributed mobile execution. Although middleware has become an essential component of modern mobile software, existing mainstream middleware mechanisms were designed with the primary focus of facilitating distributed communication and improving performance rather than on reducing energy consumption.

Network communication commonly constitutes one of the largest sources of energy consumption in mobile applications. According to a recent study, network communication consumes between 10% and 50% of the total energy budget of a typical mobile application [14]. Many mobile applications are designed with the assumption that they will be operated over some mobile networks with a certain fixed bandwidth/latency ratio. However, this assumption will not hold if an application is operated across a variety of mobile networks (WiFi, 3G, and 4G), whose conditions (e.g., bandwidth, delay, packet loss, etc) often fluctuate continuously. As an example, during the same execution, the application can be accessing a remote service using either the 3G network (low bandwidth, long delay) or the WiFi (high bandwidth, short delay). Furthermore, the conditions of either network can be fluctuating at runtime. Networks and their conditions can significantly affect how much energy is consumed by a mobile application.

Since middleware defines the patterns through which a distributed application transmits data across the network, the choice of middleware can heavily impact the amount of energy consumed by mobile applications. However, the execution patterns in mainstream middleware mechanisms are fixed; they cannot be flexibly adapted to reduce energy consumption when mobile applications switch between mobile networks with different conditions [11]. Furthermore, to maximize energy savings, the middleware execution patterns must be individually tailored for specific applications, so as to take into consideration their application logic. To support that kind of customization, middleware must be equipped with appropriate programming abstractions that can express how energy optimization strategies should be triggered and parameterized.

In this paper, we present a new middleware architecture, which equips mobile application developers with pragmatic tools and methodologies to engineer energy-efficient distributed mobile software. Our middleware architecture employs dynamic, adaptive optimization as a mechanism to minimize the amount of energy consumed by mobile applications to perform distributed interactions. We call our novel middleware mechanism e-ADAM (**e**nergy-**A**ware **D**ynamic **A**daptive **M**iddleware). e-ADAM enables the programmer to express a rich set of middleware energy optimizations and the runtime conditions under which these optimizations should be applied. e-ADAM then dynamically applies the expressed optimizations as specified for the network conditions in place.

For the thoughtful system designer, e-ADAM opens up new energy optimization opportunities at the cost of slightly increasing the programming effort: specialized optimization strategies are crafted for individual runtime conditions. However, the e-ADAM continuous dynamic adaptation makes it possible to reach the middleware energy efficiency levels that cannot be achieved via automatic optimizations performed outside of the programmer's purview.

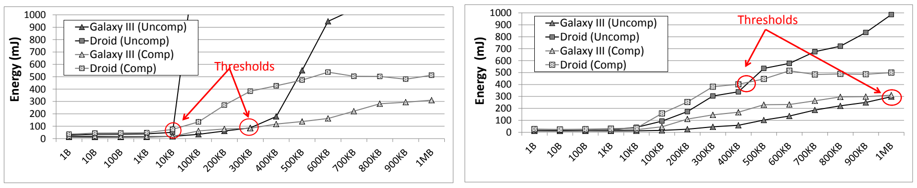
Our experiments have demonstrated the effectiveness of the e-ADAM approach to reduce the amount of energy consumed by a set of benchmarks and third-party Android applications executed across volatile mobile networks. By presenting e-ADAM, this paper makes the following technical contributions:

- **A middleware architecture that enables dynamic, application-specific energy consumption optimization:** e-ADAM features configurable energy optimization that effectively addresses the execution volatility common in modern mobile networks.
- **Application-specific energy consumption estimation:** e-ADAM features an application-level energy model that enables the e-ADAM runtime system to accurately measure and predict the energy consumption levels experienced by mobile applications under fluctuating runtime conditions.
- **Systematic evaluation:** e-ADAM optimized the amount of energy consumed by a set of benchmarks and third-party mobile applications, with the resulting energy savings as high as 30% in some cases.

The rest of this paper is structured as follows. Section 2 defines the problem that our approach aims at solving and introduces the concepts and technologies used in this work. Section 3 details our technical approach. Section 4 discusses how we evaluated our approach. Section 5 discusses the advantages and limitations of our approach. Section 6 compares our approach to the related state of the art. Section 7 concludes and presents future work directions.

2 Problem and Background

In this section, we outline the problem that our approach is intended to solve and the major technologies it uses.



(a) Mobile network. (b) WiFi network.

Fig. 1. Energy consumption comparison showing different thresholds.

2.1 Problems and Technical Challenges

The work presented here is motivated by an insight gained from our recent experiment that has studied the impact of distributed programming mechanisms on energy efficiency [10]. Our experiments measured the amount of energy consumed by passing varying volumes of data over networks with different conditions to infer common energy consumption patterns. Through this research, we have discovered that dissimilar networks consume different amounts of energy to transmit the same data. Thus, the amount of energy consumed on a network transmission can be minimized by employing the communication patterns tailored for given network conditions. In other words, adapting the execution behavior of middleware in response to changes of network conditions can reduce the overall energy consumption.

To elaborate on our prior results, we measured the amount of energy that can be saved by applying the common energy optimization technique of data compression. In this experiment, we used TCP sockets to transfer simple data buffers between a mobile device and a remote server. Figure 1 shows the impact of compressing the transferred data on the mobile device's energy consumption for the WiFi and two typical cellular networks. We experimented with two mobile devices that differed vastly in their respective hardware setups (i.e., Motorola Droid (low-end) and Samsung Galaxy S3 (high-end)) to ensure that the observed energy consumption differences are due to the network transmission rather than any other execution parameters.

When executing over the WiFi network using a high-end device, data compression does not seem to affect the amount of consumed energy. When executing over the WiFi network using a low-end device, data compression does not affect the amount of consumed energy until the 400kB data transfer threshold has been reached. Starting from that threshold, data compression ends up reducing the overall energy consumption. When executing over the 3G network and 4G network using either low-end or high-end device, data compression does not affect the amount of consumed energy until the 10kB and 300kB data transfer thresholds have been reached, respectively. Starting from that thresholds up, data compression ends up reducing the overall energy consumption.

The specific thresholds, devices, and network types used in this experiment are immaterial and only prove the point that compression must be applied in a device- and network-specific fashion, so as to maximize the potential energy savings. Because the network environment and device in place determine the thresholds at which compression should be engaged to reduce energy consumption, middleware should be able to turn this and other optimizations on and off at runtime as needed. This experiment demonstrates the need for adaptivity in middleware to be able to transfer data using the communication and execution patterns that match the execution environment in place.

At the same time, the middleware adaptations should be sufficiently general to benefit users using a variety of mobile devices. For example, Facebook reports that the mobile version of their application is accessed by 2,500 varieties of mobile devices [4]. Each of these devices is likely to exhibit different energy consumption patterns due to the hardware differences of the devices. Since it would be unrealistic to statically specify adaptations for each mobile device and application, we designed our approach to rely on runtime monitoring that can trigger the available adaptations as required by a given execution environment.

In this paper, we present a new middleware architecture that realizes the vision outlined above as energy-aware dynamic, adaptive middleware (e-ADAM). Enabling effective runtime adaptations with the goal of saving energy requires innovation in programming abstraction expressiveness and sophisticated runtime support. Specifically, our approach enables the programmer to implement multiple strategies for the same middleware functionality, each of which is deployed as dictated by the runtime changes in the execution environment. At execution time, e-ADAM monitors runtime network conditions and then automatically selects an appropriate energy optimization strategy provided by the programmer. Furthermore, in response to the changes of runtime network conditions, e-ADAM dynamically switches between the provided strategies.

2.2 Technical Background

Our middleware architecture combines dynamic adaptation and runtime energy consumption monitoring.

Middleware for Distributed Execution Our middleware architecture uses features from mainstream middleware mechanisms for distributed execution as building blocks. To facilitate effective reuse, we classify existing middleware architectures on two main axes: level of abstraction and network communication footprint. In terms of the level of abstraction, there are socket-, remote procedure call-, message-, and service-based platforms. In terms of the network communication footprints, they transfer data across the network in binary and text (primarily XML)-based formats. Major, widely used middleware architectures include sockets, Message Oriented Middleware (MOM), remote method invocation (RMI), and Web services. Our middleware architecture uses a proxy-based distributed execution mechanism and encodes the transferred data in binary.

Transport Layer Energy Saving Provisions The IEEE 802.11 standard codifies a power saving mode (PSM), under which the wireless network interface enters the sleep mode when idle. Other approaches have leveraged this mode to save energy. For example, reference [16] describes one such energy saving strategy that takes advantage of the prior knowledge of the application's communication patterns. This strategy employs a bandwidth throttling mechanism, implemented via a custom network protocol stack, to control the network transmission rate. Thus, adjusting application communication patterns can lengthen the wireless network interface's sleep time, thereby saving energy. This strategy has been shown effective in media streaming or large data transfer applications. The goal of our approach is to achieve similar energy saving benefits, but without modifying the standard protocol stack. By operating at the application level, our approach adapts crude-grained communication patterns, providing comparable energy saving benefits. For example, application communication patterns can be adapted to be periodic and predictable by breaking down large transmitted data into blocks or by reshaping the TCP traffic into bursts.

Dynamic Middleware Adaptation Dynamic middleware adaptations change the execution strategies at runtime to optimize the overall execution by leveraging the information provided by various system components. Dynamic adaptation has been also used to optimize energy consumption [6, 12]. The Odyssey platform [6] adapts data or computational quality to save energy consumption, so as not to exceed the available system resources. DYNAMO [12] is an another middleware platform that adapts energy optimizations across various system layers, including applications, middleware, OS, network, and hardware, to optimize both performance and energy. These energy-aware adaptations identify possible trade-offs between energy consumption and quality of service and then choose optimal energy optimizations based on runtime conditions.

e-ADAM shares the same vision with these approaches. However, as compared to these approaches, our approach aims at providing a high degree of customizability. It provides a programming model that enables programmers to implement application-specific energy optimization strategies as well as to express how these strategies should be applied at runtime.

3 Energy-Aware Dynamic Adaptive Middleware

In this section, we present e-ADAM by giving an overview of the approach and then describing its major parts.

3.1 Approach Overview

The e-ADAM approach hinges on the concept of configurable energy optimization strategies. e-ADAM provides a Java API for implementing the strategies, whose triggering and operation is specified using simple key-value configuration files (for an example, see Figure 3). By continuously monitoring the execution environment, the e-ADAM runtime system dynamically loads and applies the strategies as specified in the provided configurations. By selecting the strategies to apply at runtime in accordance with the environment in place, e-ADAM can optimize energy efficiency more effectively than static approaches.

Figure 2 presents the architectural design of e-ADAM that comprises three main components: a strategy manager, a runtime monitor, and an adaptation manager. First, the *strategy manager* reads configuration files and maps the parsed configurations to the available strategy implementations. Second, the *runtime monitor* continuously collects runtime information that includes network and hardware characteristics (e.g., delay, network connection type, CPU frequency, etc.) by leveraging the Android system

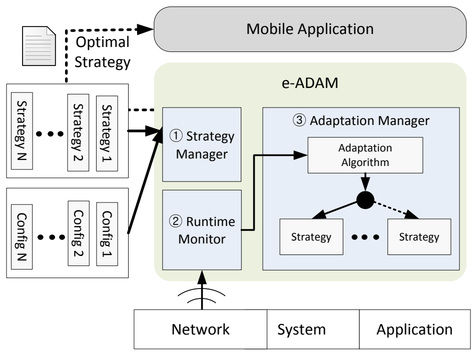


Fig. 2. e-ADAM component diagram.

APIs. Third, the *adaptation manager* correlates the collected execution data with the configuration parameters. If the resulting correlation indicates that a different energy optimization strategy should be triggered, the *adaptation manager* dynamically locates, loads, and executes the triggered strategy.

3.2 *e-ADAM* Configuration

As shown in Figure 3-(a), the e-ADAM energy optimization configuration files follow a simple key-value format, thus making it straightforward for programmers to compose and understand. Each set of configuration settings is identified by a unique name followed by a collection of key-value pairs. Configurations are demarcated by an empty line.

The three configuration keys are **execution**, **strategy**, and **criteria**. The **execution** pair identifies a remote server by means of a URL or an IP address. The **strategy** pair identifies the adaptation strategies to be applied for this communication. If a configuration file has multiple strategies, the runtime adaptation module then makes use of the selection **criteria** values. Recall that the runtime continuously applies policies speculatively, so as to evaluate their effectiveness.

The **criteria** pair defines which notion of *effectiveness* should be used with a given configuration. The **criteria** value of **energy** indicates the effectiveness to reduce energy consumption, while that of **performance** to speed up performance. The value of **epr** indicates the effectiveness to increase the energy/performance ratio. The value of **never** disables the configuration from being applied, while the value of **always** applies the configuration irrespective of its effectiveness.

```
configuration = [name]
execution = [remote API]@[address]
strategy = [name] ((and|or) [name])*
criteria = (energy|performance|
  epr|never|always)
```

(a) Configuration file format.

```
public enum Pointcut {Before, After, Around;}

public class Invocation {
  public Method getMethod() {...}
  public Object[] getParams() {...}
  public Object getResult() {...}
  public void setParams(Object[] o) {...}
  public void setResult(Object res) {...}
  public Object proceed(int mode) {...} }

public class [name] extends Strategy {
  public Pointcut getPointcut() {...}
  public Object invoke(Invocation invoc) {...} }
```

(b) Strategy API class.

Fig. 3. Energy optimization configurations.

Adaptation strategies are implemented by extending class `Strategy`, which provides a single method `invoke`. To enable the programmer to control at which execution point a strategy should be applied, e-ADAM features an Aspect Oriented Programming [8] abstraction to specify whether the implemented strategy is to be invoked `before`, `after` or `around` (instead of) a remote communication.

The components implementing the strategies referenced in configuration files follow the Java naming convention, in which class names are prefixed with their full package names (e.g., `edu.vt.eadam.Compression`). The e-ADAM runtime calls method `invoke(...)` at the specified pointcut when both the specified remote API is invoked and the energy optimization strategy is activated. A typical adaptation strategy makes use of common energy optimizations, including data compression, reducing image quality, and redirecting to an easier-to-reach remote server as discussed in Section 3.4.

3.3 *e-ADAM* Process Flow

Having described the individual components of e-ADAM, we now explain how they interact with each other. The e-ADAM process flow comprises three main processes: (1) energy consumption prediction, (2) communication monitoring, and (3) distributed communication.

The *energy consumption prediction* process estimates future energy consumption levels and communication latencies to select the specified energy optimization strategy. To that end, the adaptation manager requests snapshots of the current and prior execution environment (e.g., CPU, delay, transferred data size, execution time, etc.) from the runtime monitor and the execution history (cache), respectively. Based on these parameters, the adaptation manager estimates the energy and latency to be incurred by a given communication operation and applies the energy optimization strategy as guided by the configuration in place.

The *distributed communication monitoring* process continuously collects runtime data and creates an execution history cache to consult when estimating the energy consumption and latency of future communication operations. This process dispatches remote operations in accordance with the applied energy optimization strategy. Next, we describe each process of e-ADAM in detail.

Energy Consumption Estimation The energy consumption estimation module predicts how much energy will be consumed by a given remote communication by computing the workload expected to be carried out by the communication. Specifically, e-ADAM only computes the energy consumed by the CPU and network communication as follows:

$$\begin{aligned}\boldsymbol{E} &= E_{cpu} + E_{net} = (P_{cpu} \times T_{cpu} + P_{net} \times T_{net}) \times V \\ &= \{\Sigma(C^{act}_{cpu_f} \times T^{(u+s)}_{cpu}) + (C^{act}_{net} \times T^{act}_{net}) + (C^{idle}_{net} \times T^{idle}_{net})\} \times V\end{aligned}$$

where $C^{act}_{cpu_f}$ is the electric current of the CPU at a particular clock speed. 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. T^{u}_{cpu} and T^{s}_{cpu} are user and system times taken by the distributed execution, and they are obtained by consulting the statistics provided by the operating system (e.g., `/proc/[pid]/stat`). V is current voltage, which is also obtained from the operating system (`/sys/class/..../voltage_now`). C^{act}_{net} and C^{idle}_{net} are the electric current of the network processor required during the active and idle phases, respectively. T^{act}_{net} and T^{idle}_{net} are the active and idle runtime periods during the remote communication, respectively. These device- and execution-specific values are cached to estimate the amount of energy to be consumed during future remote communications.

Training-Based Energy Consumption Prediction To predict the amount of energy that is likely to be consumed during remote communications, e-ADAM correlates the device- and execution-specific values that were previously measured and cached. These measured values are cached and used for predicting the future energy consumption and execution time. Then, using the cached prior execution parameters (e.g., delay, communication time, transferred data size, total execution time, etc.) and the current measured execution parameters, e-ADAM predicts the communication latency. During the initialization phase, e-ADAM bootstraps the training process by executing all the strategies specified in the input configuration file and persists the obtained results to permanent storage. Based on the estimated communication latencies, as observed from prior executions, the e-ADAM runtime system predicts the expected energy consumption for a given remote communication as follows:

$$\boldsymbol{E_{prd}} = \{E_{cpu}^{avg} + (P_{net}^{act} \times T_{net}^{prd\_act}) + (P_{net}^{idle} \times T_{net}^{prd\_idle})\} \times V$$

where E_{prd} is the predicted future energy consumption, E_{cpu}^{avg} is the average energy consumption of the given remote communication, and $T_{net}^{prd\_act}$ is the predicted communication time, which are computed by correlating past execution data (e.g., delay, communication time, transferred data size, total execution time, etc), the current data size to be sent and the current network delay, respectively. The current network delay is measured by sending a probe packet and then, to avoid a delay spike, the delay value is recomputed by weighting the most recently obtained value (i.e., $delay = delay \times \alpha + delay \times (1 - \alpha)$). α was set to 0.3 in our reference implementation). The computed energy consumption value and execution time are used as a parameter for selecting the optimal energy optimization strategy for a given scenario.

Energy Optimization Strategy Selection Figure 4 shows the procedure for selecting the provided energy optimization strategies at runtime. To select an appropriate energy optimization strategy, the adaptation module predicts the future energy consumption and the future execution time by analyzing the collected runtime execution values and cached prior executions. Then, the module selects the optimization strategy that would yield either the lowest expected energy consumption, or the shortest expected execution time, or the highest expected energy/performance, as specified by given selection criteria—`energy`, `performance` `epr`, etc. While the first two parameters are self-explanatory, `epr` (the en-

```
FOREACH $strategy \in \forall Strategies$ DO
  CASE Energy:
    $E_{exptd}$ $\leftarrow$ estimateEnergy(..., $strategy$)
    IF $E_{exptd}$ is the smallest THEN
      $bestStrategy$ $\leftarrow$ $strategy$ END IF
  CASE Performance:
    $T_{exptd}$ $\leftarrow$ estimateExecTime(..., $strategy$)
    IF $T_{exptd}$ is the smallest THEN
      $bestStrategy$ $\leftarrow$ $strategy$ END IF
  CASE EPR:
    $E_{exptd}$ $\leftarrow$ estimateEnergy(..., $strategy$)
    $T_{exptd}$ $\leftarrow$ estimateExecTime(..., $strategy$)
    $epr$ $\leftarrow$ getEPR($E_{exptd}, T_{expted}$)
    IF $epr$ is the smallest THEN
      $bestStrategy$ $\leftarrow$ $strategy$ END IF
END FOREACH

$bestStrategy$.invoke(...) //Execute

/** Receive result and update exec. history */
CASE Succeed: $result$ $\leftarrow$ executionCompleted()
CASE Fail: $result$ $\leftarrow$ executionFailed()
update($result$)
```

Fig. 4. The procedure to select a strategy.

ergy/performance ratio) is a parameter that we have formulated in our prior research [10]. This ratio correlates performance and energy consumption values so as to maximize the resulting correlation. The runtime system computes `epr` value as follows:

$$EPR(x) = \frac{MIN(T_{prd}(1),...,T_{prd}(n))/T_{prd}(x)}{E_{prd}(x)/MIN(E_{prd}(1),...,E_{prd}(n))} \times 100$$

where, T_{prd} and E_{prd} are the expected execution time and energy consumption values, respectively. A higher EPR represents a condition under which the energy optimization strategy in place consumes less energy while retaining high performance as compared to other strategies.

3.4 Energy Optimization Strategies

Recall that the observation that underlies the design of e-ADAM is that certain pieces of middleware functionality can be implemented differently. In the following discussion, we give specific examples of middleware functionality and the alternatives for their implementations.

Data compression: Data compression will reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Transmitting raw data will increase network transfer, but will require less CPU processing. Which of the strategies will consume less energy depends on the runtime conditions in place. This strategy, thus, will determine the optimal compression point while considering the trade-off between CPU processing and network transfer.

Redirection: Another optimization is redirection. This strategy iterates through different endpoints of a distributed execution in the case of experiencing poor network conditions. For instance, when experiencing a network congestion at `a.com/foo`, `alt.a.com/foo` can be invoked instead. This strategy, thus, will find an optimal execution path, as operating over a congested network is likely to require additional energy resources.

Batching: A common middleware optimization is batching multiple distributed communications into a single bulk communication. For modern networks, whose bandwidth improvements surpass that of their latencies, transmitting data in bulk can reduce the aggregate latency. However, the degree of batching should be determined by the network conditions in place. Thus, the programmer should be able to specify which distributed communications should be batched and under which conditions.

In addition to the aforementioned general optimization strategies, one can also apply application-specific optimizations, tailored for particular application scenarios. For example, in a video conferencing application, the QoS can be traded for energy efficiency when the battery level gets below a certain threshold.

4 Evaluation

We have evaluated the effectiveness of e-ADAM by applying it to benchmarks and third-party applications.

4.1 Micro-Benchmarks

In this micro-benchmark, we compared the performance and energy consumption characteristics of XML-RPC and e-ADAM in executing a collection of remote invocations with different parameter sizes. In this benchmark, the client executes empty server methods with different parameters. This strategy isolates the energy consumed by the underlying middleware mechanism.

Experimental Setup The experimental setup for micro benchmarks includes a Motorola Droid (600MHz CPU, 256MB RAM, 802.11g, 3G) (a low-end mobile device), a Samsung Galaxy III (1.5GHz 2-core CPU, 2GB RAM, 802.11n, 4G) (a high-end mobile device), and a Dell PC (3.0GHz 4-core CPU, 8GB RAM) (the remote server). The network types are WiFi, 3G network, and 4G. For the WiFi, the following two network conditions were emulated: high-end (Network I: 2ms round trip time and 50Mbps bandwidth) and medium-end (Network II: 50ms and 1Mbps). The Droid used a 3G network (Network III: 70ms and 500Kbps), while the Galaxy III used a 4G network (Network III: 70ms and 1Mbps) [1]. Table 1 shows the device-specific values that parameterize the runtime systems of the mobile devices being tested.

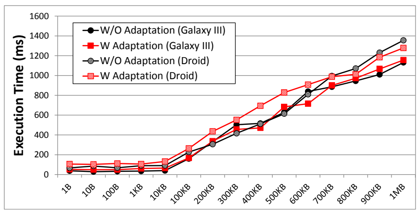
Table 1. Manufacturer provided energy profiles.

	High-end Device	**Low-end Device**		**High-end Device**	**Low-end Device**
CPU	1512.0 MHz: 577 mA	800.0 MHz: 280 mA	**WiFi**	96 mA	130 mA
	1209.6 MHz: 408 mA	685.7 MHz: 236 mA		0.3 mA	4 mA
	907.2 MHz: 249 mA	571.4 MHz: 207 mA	**Mobile**	250 mA	300 mA
	604.8 MHz: 148 mA	342.8 MHz: 165 mA		3.4 mA	3 mA
	302.4 MHz: 55 mA	228.5 MHz: 87 mA			
	N/A	114.2 MHz: 66 mA			

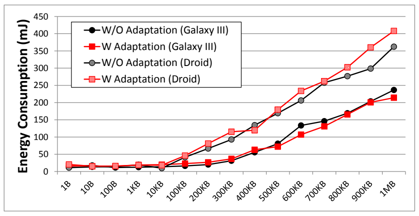
Benchmark I: Performance and Energy Consumption Overhead In this benchmark, we compared the total execution time and energy consumption of the baseline versions of the benchmarks with that using an adaptation strategy. Figure 5 (top) shows the total execution time measured on each device. As one can see, the performance overhead is quite insignificant. In particular, the overhead for both devices never exceeds 100ms and remains constant for all the measured data transfer sizes. Figure 5 (bottom) shows the amount of energy consumed by each device. As expected, the high-end device (Samsung Galaxy) consumes less energy than the low-end device (Motorola Droid). In particular, the overhead for both devices never exceeds 50mJ, which is insignificant as compared to a typical total energy budget.

Benchmark II: Adapting Energy Optimizations In this benchmark, we evaluated how the runtime system can adapt its middleware functionality between no optimization and a compression optimization in response to changes in network conditions on the high-end device. First, we evaluated how accurately the runtime system can predict how much energy will be consumed when using two different optimization strategy on the high-end device.

[1] We used Network Emulator for Windows Toolkit (NEWT) version 2.1.



(a) Total execution time.



(b) Energy consumption.

Fig. 5. Performance and energy consumption overhead.

Figure 6 shows both the predicted and the consumed energy by e-ADAM with no optimization vs. a compression optimization. In this benchmark, the average error was 23.09 % and standard deviation was 10.74 %, which is higher than in other benchmarks, whose error rates are 6-7 %). This is because when an application consumes a small amount of energy, small changes in the execution environment, such as delay or CPU frequency, can significantly affect the predicted energy consumption. (e.g., when the transferred data size increases, the average error decreases.).

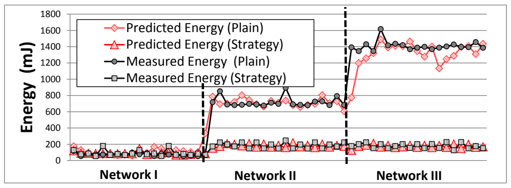


Fig. 6. Energy consumption prediction.

Then, we evaluated the effectiveness of the runtime system in selecting the energy optimization strategies that would be optimal for different execution environments. As an optimization strategy we chose data compression, which trades CPU processing for network transfer. Compressing the data reduces its size, thus reducing the workload of the remote operation transferring the data. However, running the compression algorithm uses up additional CPU cycles. First, we measured the actual amount of energy used by the same remote operation, with and without the compression strategy applied. To obtain statistically relevant measurements, each pair of remote operations (compressed and uncompressed) was repeated a 100 times under the 3 simulated networks whose parameters are explained in Section 4.1. After we measured the concrete amount of energy consumed under compression and without compression, we queried the e-ADAM runtime system whether it would trigger the compression optimization strategy. Furthermore, to evaluate the impact of the training process, we compared the effectiveness of the untrained and trained states of the runtime system (for 10 consecutive execution cycles). Table 2 shows the evaluation criteria for this experiment.

Table 2. The evaluation criteria.

Strategy causes →	**Less Energy**	**More Energy**
Trigger strategy	Success	Failure
Not trigger strategy	Failure	Success

Table 3 shows the failure rates for each network type and data size. As expected, when transferring small data volumes, compression creates some noise. Because the runtime system uses a moving average to estimate future energy consumption, it continuously reacts to the relevant changes in the execution environment. Because the runtime does not respond instantaneously, e-ADAM does not suffer from the noise that can arise due to sudden fluctuations, such as a delay spike. However, in the cases of low energy consumption (e.g. the test case with network type I and 10kB consumes only 15-100mJ.), frequent fluctuations make noise unavoidable, thus increasing the failure rate of the runtime system. However, the programmer can configure e-ADAM not to engage any optimizations when the average energy consumption level is already low. In all other test cases, nevertheless, the e-ADAM runtime system showed itself quite effective, with the training decreasing the failure rate across the board.

Table 3. Failure rate when triggering the opt. strategy.

Data Size	**Network I** Not Training / Training	**Network II** Not Training / Training	**Network III** Not Training / Training
10 kB	18 % / 12 %	9 % / 7 %	3 % / 2 %
100 kB	7 % / 2 %	3 % / 0 %	1 % / 0 %
1000 kB	3 % / 0 %	1 % / 0 %	0 % / 0 %

4.2 Case Study

To determine how well our approach works with real-world mobile applications, we experimented with open source projects, used as experimental subjects in our prior research on cloud offloading [9]. JJIL[2] is a face recognition application; its recognition functionality executes remotely in class **DetectHaarParam**. OSMAndroid[3] is a navigation application; its shortest path calculation functionality executes remotely in class **ShortestPathAlgorithm**. Mezzofanti[4] is a text recognition application; its OCR functionality executes remotely in class **OCR**.

The experimental setup for the case study includes a same high-end mobile device and a same remote server. The mobile device is connected to two emulated WiFi networks (Network I and Network II) and a cellular network (Network III).

For each subject, we measured the amount of the energy consumed and the execution time by typical, simple use cases. Specifically, for OSMAndroid, we selected two locations and the requested route between them. For the face recognition application, we examined one image file for the presence of human faces. Then, we selected the compression strategy for the face recognition application because it transfers a large amount of data; we selected the redirection strategy for OSMAndroid. The use cases were executed under two optimization modes: (1) original distributed execution without an energy optimization and (2) the e-ADAM approach with either the `epr` or `energy` criteria. Figure 7 shows the

[2] `http://code.google.com/p/jjil/`
[3] `https://code.google.com/p/osm-android`
[4] `https://code.google.com/p/mezzofanti`

configurations used in this case study. In this case study, we do not compare other middleware architectures with our approach because we already showed effectiveness of our approach in the prior benchmarks.

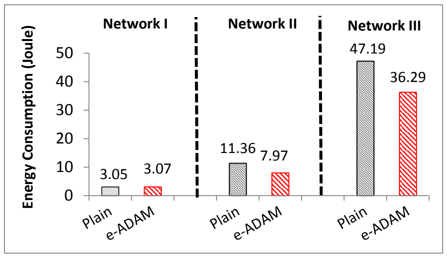
Figure 8-(a) shows how the e-ADAM approach has reduced the amount of energy consumed by the face recognition application. Because in a high-end mobile network (i.e., Network I) the compression strategy incurs additional processing overhead, e-ADAM does not apply this strategy. However, in other networks (i.e., Network II and III), the compression strategy reduced the amount of energy consumed by 30%. Figure 8-(b) shows the total execution time taken by the face recognition application. Similarly, e-ADAM improved the overall performance.

For the OSMAndroid application, we used a different scenario. Because the application transfers less data than the first subject application, we selected a redirection strategy. Figure 8-(c),(d) show the energy consumption and total execution time for the subject application. At the first phase, two remote servers (i.e., Server I and II) have the same execution environments (e.g., network condition), but at the second phase, we injected network delay to both remote servers and injected 500 ms processing delay at the Server I. With the `epr` criteria, while e-ADAM selects Server I during the first phase, it selects Server II during the second phase, as it considers both energy consumption and performance metrics when selecting an optimal strategy.

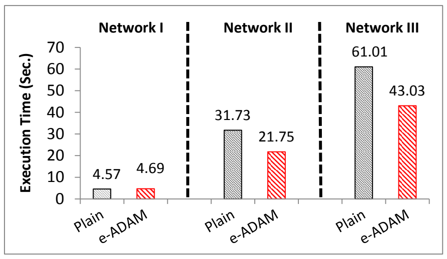
```
configuration = JJIL
  execution = DetectHaarParam.push(*)@[*:*]
  strategy = edu.vt.eadam.Compression
  criteria = energy

configuration = OSMAndroid
  execution = ShortestPath.execute(*)@[*:*]
  strategy = edu.vt.eadam.Redirection;
  criteria = epr

configuration = Mezzofanti
  execution = OCR.ImgOCRAndFilter(*)@[*:9999]
  strategy = *.Batching and *.Compression
  criteria = energy
```

Fig. 7. Configuration files.

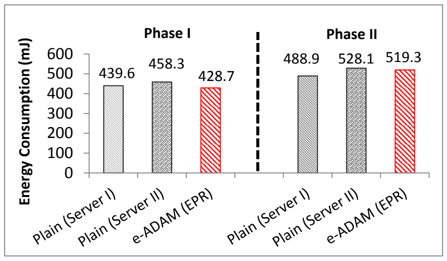
For the OCR application, we used two strategies—`Batching` and `Compression`—to optimize the transfer of the fragments of a large (∼6MB) image file. The strategies are applied sequentially in the order of appearance in the configurations, `Batching` followed by `Compression`. Figure 8-(e),(f) show the results of e-ADAM applying these strategies: first, `Batching` alone and then combined with `Compression`. In a high-end mobile network (i.e., Network I) compressing data incurs additional processing overhead, whose energy costs are not offset by the resulting reductions in bandwidth utilization. Thus, for these networks, the `Batching` strategy should be the only one applied. However, in limited networks, adding the `Compression` strategy causes the overall energy consumption to be reduced by 20%. Figure 8-(e),(f) show the total execution time and total execution time for the same OCR application. Energy consumption and total execution time are positively correlated. Indeed, e-ADAM reduced the total runtime by 6% and 14%, when the `Batching` and the (`Batching` + `Compression`) strategies were applied, respectively. Furthermore, reusing the `Compression` strategy has reduced the implementation burden of this case study.



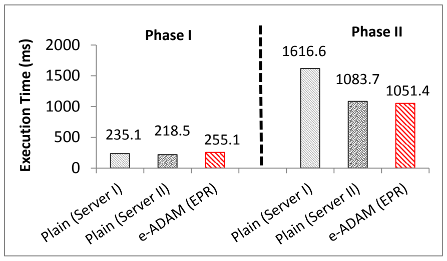
(a) Face recognition (energy).



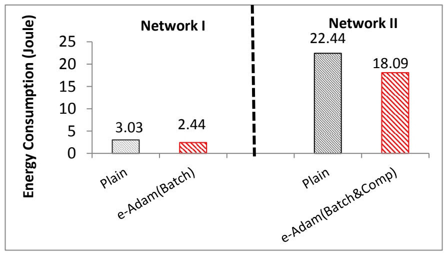
(b) Face recognition (time).



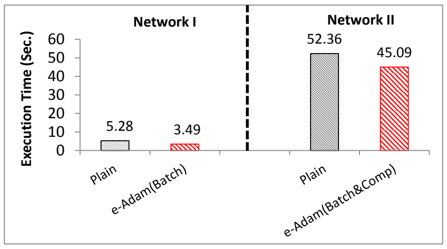
(c) OSM-Android (energy).



(d) OSM-Android (time).



(e) OCR (energy).



(f) OCR (time).

Fig. 8. Energy consumption and execution time of the subject applications.

5 Discussion

e-ADAM enables a greater separation of concerns in that it can change a mobile applications's energy/performance characteristics without affecting its core business logic. The energy optimization strategies and the configurations to apply them are expressed separately from the main source code. This degree of separation also makes it possible to effectively reuse energy optimization strategies and configurations across components and applications.

Although e-ADAM can deliver tangible benefits to the mobile application programmer, it also has some inherent limitations. In particular, the limitations concern its ranges of applicability and usability. The overhead imposed by the e-ADAM runtime makes the approach inapplicable to those distributed mobile applications that use simple, infrequent remote interactions. The runtime overhead is offset if the optimized application spends a substantial amount of energy on remote interactions. Thus, application designers have to decide whether using e-ADAM would be beneficial for each application.

Another limitation of e-ADAM is that the approach is automated rather than automatic. The programmer is responsible for implementing energy optimization strategies using the provided API and for expressing how the strategies should be applied. Although implementing common optimization strategies is facilitated by the presence of multiple third-party libraries and frameworks, the programmer must be aware of which predefined building blocks they have at their disposal.

Finally, although enhanced configurability and adaptability of communication are key to improving the energy efficiency of mobile applications, having to learn how to use a new middleware interface may negatively affect programmer productivity. Nevertheless, we argue that the design of e-ADAM eases adoption—e-ADAM can be used as a drop-in replacement for any middleware, structured around the RPC paradigm.

6 Related Work

Reducing the energy consumption of mobile applications to extend the battery life of mobile devices has been the focus of multiple complimentary research efforts, including system- and application-level optimizations. The system-level optimizations include CPU scheduling algorithms [18], disk power managements [17], network interfaces [1], specialized-network protocols [2], and process migration [3]. Although these system-level optimizations have proven quite effective in extending the battery lives of mobile devices, the system changes these optimization require complicate their deployment to heterogeneous mobile devices.

In contrast to system-level optimizations, application-level optimizations provide pragmatic, automatic tools or guidelines to the programmer [9, 13, 7]. The effectiveness of application-level optimizations hinges on the accuracy of the information provided by the underlying system and execution environments.

Cross layer optimizations leverage the information provided by multiple system layers. Odyssey orchestrates the interactions between the OS and applications [6]. Similarly to our approach, Spectra [5] provides a specialized APIs for the mobile programmer. By monitoring multiple execution environments, Spectra selects an optimal communication path to a remote server. While Spectra only provides a single fixed optimization, e-ADAM enables the programmer to implement multiple application-specific optimizations. The e-ADAM approach makes it possible to reuse known energy optimization techniques to design application-specific energy optimizations.

7 Conclusions and Future Work

This paper has presented e-ADAM, a middleware architecture that employs dynamic adaptation to reduce the energy consumption of mobile applications. e-ADAM features a sophisticated runtime system that predicts and regulates the energy consumed during remote interactions. By means of configurations, the runtime can deploy programmer-provided optimization strategies. Our evaluation comprised applying e-ADAM to reduce the energy consumed by benchmarks and third-party applications under different execution environments. These results indicate that the e-ADAM approach represents a promising direction in developing energy efficient mobile applications.

As future work, we plan to apply e-ADAM to improve the energy efficiency of cloud offloading, another energy optimization technique. Cloud offloading makes it possible to execute the application's energy intensive functionality in the cloud, thereby reducing the amount of energy consumed by the mobile device running the application [19, 9]. Integrating middleware-related optimizations with cloud offloading has the potential to open up new energy optimization opportunities for mobile applications.

Acknowledgments

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

References

1. M. Anand, E. Nightingale, and J. Flinn. Self-tuning wireless network power management. *Wireless Networks*, 11(4):451–469, 2005.
2. N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc. of the* 9th *ACM SIGCOMM Conference on Internet Measurement Conference*, 2009.
3. B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proc. of the 6th ACM European Conference on Computer Systems (EuroSys)*, 2011.
4. Facebook Mobile. Facebook for every phone, July 2011.
5. J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.
6. J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.
7. I. Giurgiu, O. Riva, and G. Alonso. Dynamic software deployment from clouds to mobile devices. In *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, 2012.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming(ECOOP 97)*, pages 220–242, 1997.
9. Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proc. of the 32nd International Conference on Distributed Computing Systems*, 2012.
10. Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, 55(9), 2013.

11. A. Miettinen and J. Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proc. of the 2nd USENIX conference on Hot Topics in Cloud Computing*, 2010.
12. 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.
13. A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenIT. In *Proc. of the 1st International Workshop on Green and Sustainable Software*, 2012.
14. A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proc. of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.
15. K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.
16. E. Tan, L. Guo, S. Chen, and X. Zhang. PSM-throttling: Minimizing energy consumption for bulk data communications in WLANs. In *Proc. of the IEEE International Conference on Network Protocols*, 2007.
17. 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.
18. 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.
19. Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.