Facilitating the Implementation of Adaptive Cloud Offloading to Improve the Energy Efficiency of Mobile Applications

Young-Woo Kwon
Department of Computer Science
Utah State University
young.kwon@usu.edu

Eli Tilevich
Department of Computer Science
Virginia Tech
tilevich@cs.vt.edu

***Abstract*—Cloud offloading—leveraging remote cloud-based computing resources to execute energy-intensive functionality—has become a common optimization technique for mobile applications. However, implementing cloud offloading techniques remains a delicate and complex task, reserved for expert programmers. If cloud computing is to realize its promise as a generally applicable, powerful optimization technique for mobile applications, its implementation barrier must be lowered. As we have discovered, reusable system building blocks exposed via a convenient programming model can facilitate the implementation of complex cloud offloading optimizations. This paper describes a system architecture for implementing adaptive cloud offloading optimizations. In particular, the architecture features parameterizable building blocks for monitoring and estimating energy consumption and performance efficiency as well as state synchronization across address spaces, which the mobile programmer can use à la carte. These blocks streamline the implementation procedure for a wide array of adaptive offloading optimizations. Applying this system architecture to third-party mobile applications has optimized their energy efficiency, depending on the execution environment in place.**

***Index Terms*—mobile applications; energy optimization; cloud offloading; programming model; adaptation**

I. Introduction

As the energy demands of modern applications continue to outstrip the battery capacities of mobile devices, the topic of energy optimization has enjoyed a wide coverage in the research literature. A particularly popular energy optimization technique is *cloud offloading*—transforming a mobile application to move its functionality containing energy spots to execute at a cloud-based server, with the parameters and results transferred across the mobile network. The energy savings afforded by cloud offloading stem from the functionality executed by cloud-based servers not draining the battery power of the mobile devices running the application.

The main difficulties of implementing effective cloud offloading optimizations lie in the high heterogeneity of mobile hardware and the volatility of mobile networks. Mobile devices come in numerous hardware configurations, which vary in terms of their respective CPU, memory, and communication characteristics. For example, Facebook reports that the mobile version of their application is accessed from more than 2,500 varieties of mobile devices [1]. The same mobile application running on devices with dissimilar hardware capacities would consume divergent amounts of energy. In addition, the mobile network characteristics in place have been shown to affect the amount of energy spent transferring the same parameters across the network [2]. Finally, mobile applications are highly context-sensitive, with different users favoring energy and performance efficiencies to varying degrees.

Hence, research in cloud offloading is steadily heading in the direction of adaptivity, deciding on what functionality should be offloaded at runtime [3]. Unfortunately, implementing effective adaptive cloud offloading optimization is non-trivial. Although cloud offloading techniques differ widely in their respective designs, all need the ability to make informed decisions whether and when an offloading is most likely to save energy. Also, when offloading a functionality without modifying the underlying runtime system, the program state must be correctly and efficiently synchronized across the mobile device and cloud server. Implementing these mechanisms can incur large engineering overheads, slowing research progress and rendering the technique prohibitively complex for real-world environments.

Having studied cloud offloading for the last couple of years, we have discovered that the implementation barrier for this powerful energy optimization can be effectively lowered if its main building blocks can be provided as a general system architecture exposed to the programmer via a convenient programming model. In particular, we argue that a system architecture featuring four main building blocks—(1) energy consumption monitoring and estimation, (2) performance efficiency monitoring and estimation, (3) program state synchronization, and (4) the offloading process monitoring and notification—can be effectively leveraged to straightforwardly implement adaptive cloud offloading optimizations. We show how these building blocks enable a flexible system architecture for enhancing third-party mobile applications with the ability to adaptively optimize their energy efficiency.

By presenting our novel system architecture for optimizing the energy efficiency of mobile applications, this paper makes the following contributions:

978-0-7695-5566-9/15 $31.00 © 2015 IEEE
DOI 10.1109/MobileSoft.2015.21

- An extensible system architecture for implementing cloud offloading optimizations with reusable building blocks that the programmer can use à la carte.
- A programming model for using the building blocks as required to implement adaptive offloading optimizations.
- A case study of using the system architecture to engineer a non-trivial cloud offloading optimization; the engineered optimization was applied successfully to reduce the energy usage of 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. Section III details our technical approach. Section IV and Section V discuss how we evaluated our approach. Section VII compares our approach to the related state of the art. Section VI discusses the advantages and limitations of our approach, and Section VIII concludes this paper.

II. Problem Definition and Technical Background

In this section, we define the problem to be solved and provide a technical background for the main technologies used in our solution.

A. Motivation and Problem Definition

Having studied the cloud offloading optimization in the last couple of years, we have found that the heterogeneity of mobile hardware and volatility of mobile networks require highly adaptive energy optimizations to be effective. However, implementing adaptive optimizations can be a daunting engineering undertaking, suited only for expert programmers. The goal of this research is to define a system architecture that features reusable building blocks that can be used to streamline the process of implementing adaptive cloud offloading optimizations.

Figure 1 demonstrates the need for adaptivity in mobile applications as compared to applications written to run on stationary computers. When it comes to energy efficiency, the more diverse the execution environment, the more configurable the software that runs in it needs to be. Desktop applications tend to have fixed configurations that reflect the end user's preferences mostly with respect to the application's look-and-feel. Laptops can be configured at the system level to save energy when battery-powered; application-level energy usage options are not common. Finally, applications written for tablets and smartphones commonly feature per-application energy saving options. Mobile applications can be configured to behave differently depending on the mobile network in place. For example, the user can configure to compress the data transferred only when no Wi-Fi network is available.

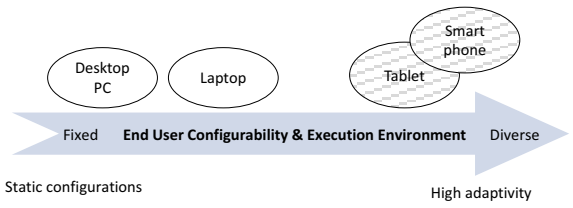


Fig. 1. Characteristics of mobile software.

Thus, an effective energy optimization strategy for mobile applications should take into consideration the device's hardware characteristics, the execution environment in place, and end-user preferences. Properly accounting for all these variables requires that optimizations be highly adaptive. Implementing adaptive behaviors, however, can be prohibitively complex from both the development and maintenance perspectives, especially for application programmers. This paper advocates a system architecture that can simplify the implementation of adaptive energy optimization by providing reusable building blocks.

B. Technical Background

The technical concepts behind our approach include cloud offloading and system-level adaptation. We describe these technologies in turn next.

1) Cloud Offloading: Cloud offloading optimizes the energy consumption of mobile applications by executing the application's energy intensive functionality in the cloud, so as to avoid draining the mobile device's battery power. A typical implementation technique for cloud offloading is program partitioning, transforming the original mobile application into: the client part running on a mobile device and the server part running in the cloud; all the calls between the parts become network communication. Program-level cloud offloading is an application of automated program partitioning, an automated technique for transforming centralized applications to run across the network using a compiler-based tool [4]. In this work, we focus on method-level offloading, in which application functionality is shipped to be executed by the server at method granularity. This design assumption is in line with solid object-oriented design principles, which favor fine-grained methods. If the functionality to offload is not confined to method boundaries, an Extract Method refactoring [5] can be performed to restructure the code, making it amenable to method-level offloading. Representative recent cloud offloading approaches leverage program analysis, automated transformation, and adaptive runtime [6], [7].

In our prior work, we introduced cloud offloading that transforms applications, leveraging static program analysis and program transformation techniques, without destroying their ability to execute locally in the case of network disconnection [8]. Then, we introduced a cloud offloading technique that takes advantage of dynamic adaptation by means of a hand-crafted runtime system [6]. Specifically, this technique automatically enhances a centralized program with the ability to execute across the network, while the local/remote parts are determined dynamically at runtime, as required by the current execution environment. The approach presented in this paper draws on the lessons of these prior approaches by innovating in the implementation design space.

2) System-Level Adaptation: Middleware can be adapted at runtime to change its execution strategies, so as to optimize the overall distributed execution in accordance with the information received from various system components. Researchers have leveraged dynamic adaptation to optimize energy consumption [9], [10]. The Odyssey platform [9] provides the ability to adapt data or computational quality as a mechanism for reducing energy consumption and operating within the available system resources. Another middleware platform adapting energy optimizations across system layers is DYNAMO [10]; it optimizes both performance and energy by adapting at the level of applications, middleware, OS, network, and hardware. By identifying possible trade-offs between energy consumption and QoS, these energy-aware adaptations can choose optimal energy optimizations for a given runtime condition.

The approach presented here shares the system design objectives with these approaches. The novelty of our approach lies in its system implementation benefits. Our design goal is to enable state-of-the-practice developers to take advantage of adaptive energy optimizations implemented as cloud offloading. We advocate a programming model that enables programmers to implement customizable cloud offloading optimizations as well as to express how the runtime system should be tailored for given end-user preferences.

III. Reusable Building Blocks for Adaptive Energy Optimization

The key contribution of this work is the analysis and identification of the engineering issues that arise when realizing adaptive cloud offloading mechanisms, which have become increasingly prevalent in the mobile energy optimization design space. In this section, we present our system architecture that offers reusable building blocks that can be used to construct adaptive cloud offloading optimizations. We start by giving an overview of the general system architecture and then describe its major implementation blocks, with a particular emphasis on the programming model. We discuss the approach's applicability and limitations in Section VI.

A. System Architecture

Figure 2 gives an overview of our system architecture. Our primary design intent is to simplify the implementation of adaptive cloud offloading energy consumption optimizations. Since there is a great variety of offloading schemes, the architecture provides reusable implementation blocks that can be accessed through a convenient programming interface. When implementing their cloud offloading optimizations, programmers are free to choose any offloading strategies and algorithms. One design assumption is that the offloading will be performed at a method-level, even though the number of offloaded methods at a time is flexible. Our architecture provides reusable blocks to monitor and estimate the amount of energy consumed and execution time taken by a given method or a set of methods. Another set of blocks provides

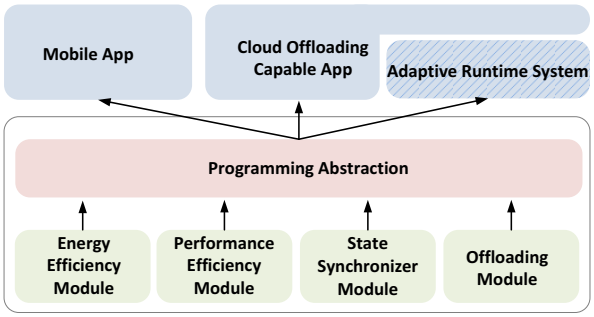


Fig. 2. Overview of the system architecture.

reusable functionality to synchronize program state between the client memory space and that of the offloading server.

B. Programming Model

The programming model described in Figure 3 features five Java classes (four concrete and one abstract) and two interfaces. The `EnergyConsumptionInfo` and `PerformanceEfficiencyInfo` classes provide functionality to measure and estimate energy consumption and performance efficiency for a given method, respectively. To reduce overhead and to avoid conflicts with accessing OS-level resources, these classes are implemented as singletons—not more than one object of each type ever instantiated per a mobile application.

Class `StateSynchronizer` reconciles program states after an offloading operation is completed. It uses the `copy-restore` semantics [11], which efficiently updates object graphs while preserving their aliasing semantics (i.e., referring to an updated object using multiple references). To use the services of class `StateSynchronizer`, object graphs have to be represented as `State` objects.

The abstract class `OffloadingRuntime` should be implemented by the programmer by providing the functionality of method `offload`. Because mobile applications are known to use a variety of communication mechanisms, our design leaves this method entirely to the programmer's purview. Whether an offloading operation completes successfully or experiences a failure, the concrete implementations of the class call methods `offloadingCompleted` or `offloadingFailed`, respectively. By extending class `Observable`, `OffloadingRuntime` implements the Observer/Observable design pattern [5], so that any object whose class implements JDK interface `Observer` can register to have its method `update` invoked. If the offloading succeeds, the method's parameter will be the updated `State` object; if it fails, the parameter will be an `Exception` object describing the failure's reason.

Having introduced the programming model of our system architecture for adaptive cloud offloading, we next provide the implementation details of our system infrastructure exposed through this model.

```
public class EnergyConsumptionInfo {
  public static EnergyConsumptionInfo create() {...}
  public void startEnergyMeasurement(Method m) {...}
  public double endEnergyMeasurement(Method m) {...}
  public double estimateEnergy(Method m, Object[] p, URI h) {...}
  public void addListener(IEnergyListener l, long time) {...}
}


public interface IEnergyListener {
  public void notifyConsumedEnergy(double energy) ;
}
```

(a) Energy efficiency API and interface.

```
public class PerformanceEfficiencyInfo {
  public static PerformanceEfficiencyInfo create() {...}
  public void startPerfMeasurement(Method m) {...}
  public long endPerfMeasurement(Method m) {...}
  public long estimateExecTime(
         Method m, Object[] params, URI host) {...}
  public int[] checkDelays(Set<URI> hosts) {...}
  public void addListener(IPerfListener l, long time) {...} }

public interface IPerfListener {
  public void notifyTakenTime(long time) ;
}
```

(b) Performance efficiency API and and interface.

```
public class StateSynchronizer {
  public void synchronize(State newState, State oldState)
  {...}
}

public class State {
  ...
  public void addObject(String key, Object obj) {...}
  public void removeObject(String key, Object obj) {...}
  public Object getObject(String key) {...}
}
```

(c) State synchronization API.

```
/** To be notified when an offloading completes successfully or fails ;
   classes can implement and add themselves as Observer for this class . */
public abstract class OffloadingRuntime extends Observable {
  public abstract void offload(State toServer);

  public void offloadingCompleted(State fromServer) {
    notifyObservers(fromServer);
  }
  public void offloadingFailed(Exception e) {
    notifyObservers(e);
  } }
```

(d) Observable offloading runtime class.

Fig. 3. Programming model and provided interfaces.

C. Runtime Implementation

1) Energy and Performance Efficiency: As explained above, an important piece of functionality of our system architecture is estimating the amount of energy that would be consumed if a given method is to be offloaded. To that end, when the `estimateEnergy` method in class `EnergyConsumptionInfo` is invoked, the runtime system computes the results as follows:

$$\boldsymbol{E} = \{\Sigma(P^{act}_{cpu@f} \times T^{(u+s)}_{cpu}) + (P^{act}_{net} \times T^{act}_{net}) + (P^{idle}_{net} \times T^{idle}_{net}\ \})$$

where $P^{act}_{cpu@f}$ is the power that the CPU consumes at a given clock speed. Modern CPUs provide speed-step, an OS facility to dynamically change the processor's clock speed, in which each clock speed leads to different levels of power consumption.

T^{u}_{cpu} and T^{s}_{cpu} are process-specific user and system times, respectively. P^{act}_{net} and P^{idle}_{net} are the required power by the network processor during the active and idle phases, respectively. T^{act}_{net} and T^{idle}_{net} are the active and idle portions of the remote communication, respectively.

The runtime system uses the same parameters when computing the result for the `estimateExecTime` method in class `PerformanceEfficiencyInfo` as follows:

$$\boldsymbol{T^{exptd}_{net}} = Avg(\Sigma T^{idle}_{net}) + T^{exptd}_{net\_act}$$

where T^{exptd}_{net}—the future execution time averages the idling time and the expected communication time periods during a given measurement window; the bigger the window, the more accurate the estimate.

Figure 4 shows how we measure the delay and each time taken during the idling, sending and receiving phases. These times are averaged in a given measurement window to estimate the expected communication time. The estimates are cached to be used to predict the communication time and execution time.

Our energy prediction takes into account the CPU and network energy consumptions. The CPU energy consumption is predicted by averaging previous executions. The network energy consumption is predicted by extrapolating the average between the last two measured energy consumption values as depicted in Fig 5. To that end, the runtime system uses the cached execution parameters (e.g., delay, communication time, transferred data size, total execution time, etc.) and the current execution parameters to predict the expected communication time period during a given measurement window as follows:

$$\boldsymbol{T^{exptd}_{net\_act}} = \{T[i-1] + T[i-1] \times \tfrac{\Delta L(i,i+1)}{\Delta L(i-1,i+1)}\} \times \tfrac{\Delta D(i,i+1)}{\Delta D(i-1,i+1)}$$

where T, L, and D denote the size of the program state to be transferred for each remote execution, the current network delay, the cached communication time, respectively. Because predicting future energy consumption levels and execution time utilizes prior execution history, these device- and execution-specific values are cached to compute the amount of energy to be consumed and the time to be taken during potential offloading operations.

2) State Synchronization: Another important functionality of our system architecture is synchronizing the program state across different address spaces. The program state is synchronized by means of copy-restore, an advanced semantics for passing linked data structure as parameters to remote methods (e.g., linked lists, trees, and maps) [11]. The copy-restore semantics copies all reachable parameter state to the server and then overwrites the client's state with the server modified data in-place, so that all the client-side aliases are preserved.

3) Implementing Offloading Optimizations: Since offloading strategies come in great varieties, it is hard to design

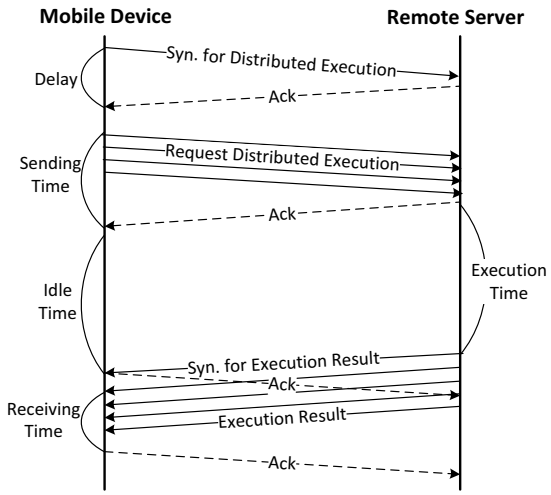


Fig. 4. Measuring communication time.

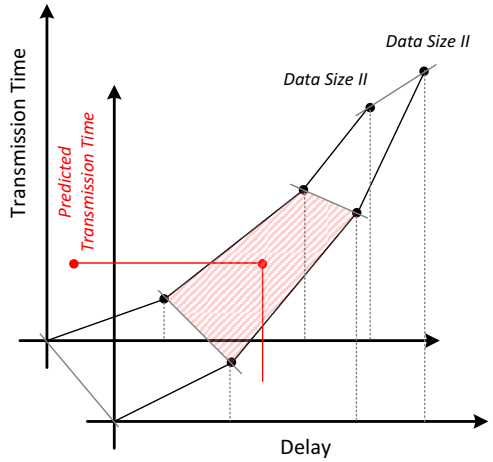


Fig. 5. Estimating communication time.

reusable implementation blocks that can benefit all of them. In particular, application-level offloading strategies differ in terms of the middleware they use to communicate between the mobile device and the server (e.g., sockets, messaging, synchronous/asynchronous remote calls, etc). Irrespective of the communication middleware used, the programmer may also choose to run the offloading task as a separate execution thread, enabling the user to continue interacting with the mobile application while an offloading is in progress.

The only two common pieces of functionality for different offloading mechanisms is that an offloading needs to start, passing some program state to the server, and then to complete, receiving some program state from the server. The passed program state can then be integrated with the existing state. An offloading can also fail, a condition reported by the Java runtime system by throwing an object whose class is derived from `Exception`.

To support both synchronous and asynchronous offloading strategies, our system architecture uses the Observer/Observable design pattern to notify any relevant objects when an offloading either completes successfully or fails. For example, both the state synchronization component and the monitoring component may need to be notified when an offloading is successfully completed to synchronize the received state and to update the execution state, respectively. This pattern enables a loose coupling between the runtime object and its observer objects, thus making it possible to support a wide variety of offloading strategies.

IV. Case Study

In this section, we describe a case study of applying the system architecture and programming model described above to implement a state-of-the-art adaptive offloading mechanisms. The purpose of this case study is to validate whether our architecture and model can be used effectively to engineer a practical adaptive cloud offloading mechanism. The mechanism implemented here introduces cloud offloading with the ability to configure the offloading mechanism with user-specific preferences for energy and performance. The mechanism implemented here extends our prior work on cloud offloading [6] with the ability to configure the offloading mechanism with user-specific preferences for energy and performance.

A. Development Process

Figure 6 shows the workflow we followed to implement an adaptive cloud offloading strategy that leverages the system architecture described in Section III. The programmer first specifies the methods that were found to be energy hotspots. This information can be easily obtained by energy profiling the application, a development phase that is orthogonal to our implementation. The programmer also provides a configuration file that parameterizes our runtime system with the criteria that should be optimized when making offloading decisions. The user input is verified and the program's source code is automatically enhanced to generate checkpoints saving the necessary program state for offloading any portion of the specified energy hotspots. Due to its application specificity, this program transformation cannot be made into a reusable component.

For this implementation, we needed an adaptive runtime system that satisfies the following set of requirements. Multiple offloading servers connected to the device with mobile networks with different latency/bandwidth characteristics; a configuration file that specifies whether energy or performance should be favored when planning offloadings.

To make the runtime system configurable, the implementation will provide a configuration file that can be used to specify user preferences with respect to the offloading optimizations. Figure 7 shows the syntax that must be followed by the configuration files used with this runtime system.

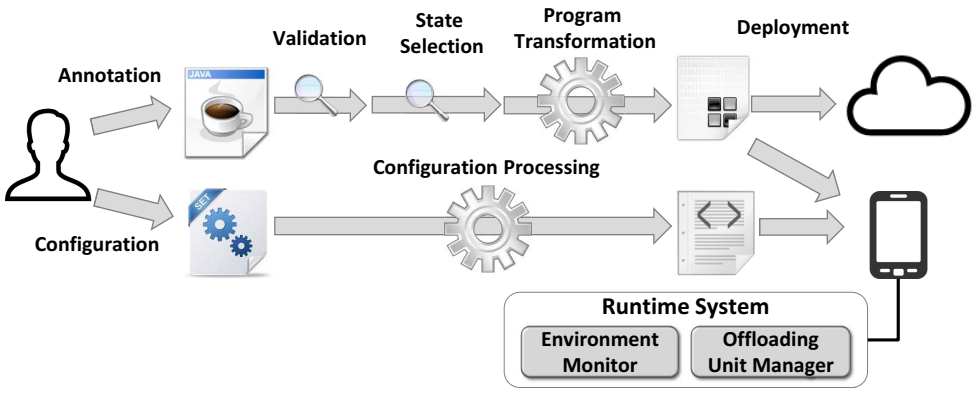


Fig. 6. Workflow to implement adaptive cloud offloading.

```
hotspot=[methodName]
host=[url]+
mode=[plain|adaptive]
criteria=[energy|performance|epr]
strategy=[name]
```

Fig. 7. Configuration file syntax.

A configuration file contains a set of key/value pairs, with the keys of `hotspot`, `host`, `mode`, and `criteria`. The `hotspot` key points to the method identified as an energy hotspot. The `host` key points to the locations of the available offloading servers. The `mode` key points to the value specifying whether the offloading should be plain (i.e., always offload the specified hotspot method) or adaptive (i.e., decide whether to offload at runtime based on the conditions in place). The `criteria` key defines which notion of *effectiveness* should be used with a given offloading. 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 that correlates performance and energy consumption values so as to maximize the resulting correlation. The `strategy` key points to well-known energy optimization techniques, including data compression, reducing image quality, and redirecting to an easier-to-reach remote server.

Based on the requirements for the cloud offloading implementation explained above, a runtime system satisfying them can be implemented using our system architecture as shown in the component diagram in Figure 8. The main components of the runtime system are the configuration handler, the adaptation module, and the network module. The configuration handler instantiates offloading plans as specified in the configuration files it parses. The adaptation module makes offloading decisions; it makes use of the reusable energy efficiency and performance efficiency building blocks. The network module

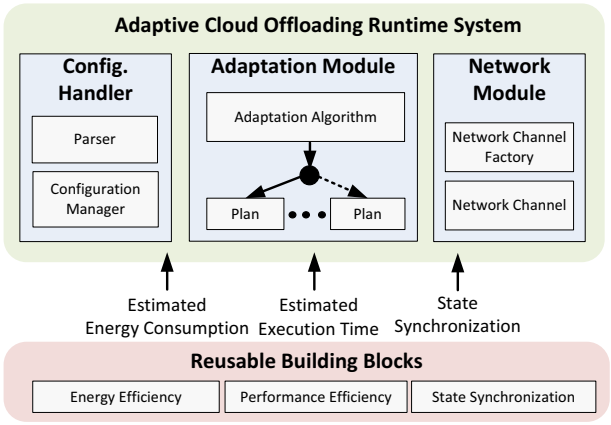


Fig. 8. Adaptive cloud offloading runtime system.

communicates with the offloading servers; it makes use of the reusable state synchronization building block.

The purpose of this case study is to determine whether the reusable blocks of our system architecture provide sufficient functionality to implement a runtime system that can support a non-trivial adaptive cloud offloading mechanism.

B. Supporting Configurability

Figure 9 shows a flowchart of the implemented adaptive runtime system and corresponding pseudo code. The figure shows how the API of our programming model can be used to implement the runtime logic of this particular offloading mechanism. To identify the most suitable offloading server (i.e., the one fitting the specified user preferences), the runtime system obtains the expected energy consumption and execution time for each available server using the provided APIs. In the end, the runtime system selects the server offloading to which would yield either (1) the lowest energy consumption, (2) the

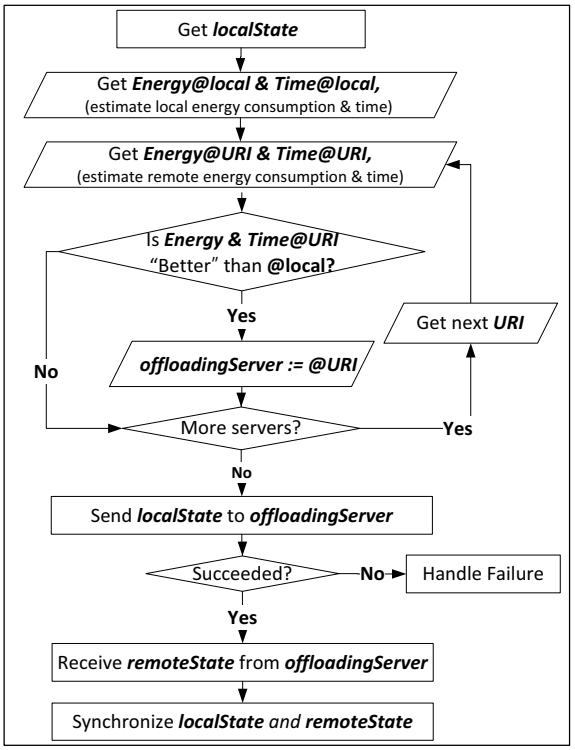


Fig. 9. Flowchart of the case study.

shortest execution time, or (3) the highest energy/performance ratio, as specified by a given configuration file (i.e., the `criteria` options of `energy`, `performance`, and `epr`).

Since mobile applications are known to use different versions of communication mechanisms to exchange data with remote servers, our system architecture does not provide any communication building blocks. However, it does provide a fine-tuned implementation of an algorithm that synchronizes objects graphs in place [11]. By simply calling a provided API method with the original and updated program states, the programmer can efficiently replay the changes made to the updated state on the original state, while keeping all the aliases pointing to the objects of the original state intact. This hard-to-implement functionality is essential, when implementing offloading mechanisms for applications written in modern managed languages such as Java and C#.

V. Evaluation

We have evaluated the effectiveness of the cloud offloading mechanism constructed in the case study in Section IV by applying it to third-party Android applications. In our implementation, we deliberately avoided putting in place any fine-grained performance optimizations, thus focusing on the architectural effectiveness of our programming model.

A. Programmability

First, we evaluated the software engineering benefits of programming model. To that end, we compared two different

```
FOREACH uri ∈ ∀server DO
  CASE Energy
    E_exptd ← estimateEnergy(..., uri)
    IF E_exptd is the smallest THEN server ← uri END IF
  CASE Performance
    T_exptd ← estimateExecTime(..., uri)
    IF T_exptd is the smallest THEN server ← uri END IF
  CASE EPR
    E_exptd ← estimateEnergy(..., uri)
    T_exptd ← estimateExecTime(..., uri)
    epr ← getEPR(E_exptd, T_expted)
    IF epr is the smallest THEN server ← uri END IF
END FOREACH

/** send toServer state to server, the selected server */
sendToServer(server, toServer)

/** receive fromServer state or exception and
    notify it all registered observers */
CASE Succeed
  fromServer ← offloadingCompleted()
  update(fromServer)
  /** synchronize the remote and local program states */
  synchronize(fromServer, toServer)
CASE Fail
  exception ← offloadingFailed()
  update(exception)
```

Fig. 10. Pseudo code of the case study.

implementations of the same adaptive offloading mechanism: original with all the functionality implemented from scratch and framework-based with the major functionality provided by our framework. In Table I, for each implementation, we report the total lines of uncommented code (LOC) and the code's McCabe Cyclomatic Metric (MCC), a standard measure of code complexity[1].

TABLE I
Expressiveness Comparison.

Approach	LOC	Max. MCC
Original Impl.	14,089	89
Framework-Based Impl.	189	6

As expected, it takes two orders of magnitude fewer lines of code to implement the mechanism using our framework than coding all the required functionality from scratch. Similarly, the programming effort as measured by the MCC metrics is also much lower for the framework-based implementation. Although reducing the programming effort is the raison d'être of programming libraries and frameworks, it is still important to quantify the software engineering benefits afforded by our framework.

B. Experimental Setup

The experimental setup includes a mobile device (1.5GHz dual-core CPU, 2GB RAM, 802.11n) and a remote server (3.0GHz quad-core CPU, 8GB RAM). The network types are two emulated networks: high-end (WiFi network: 20ms round trip time (RTT) and 50Mbps bandwidth) and low-end (4G

[1]We used Metric 1.3.6 http://metrics.sourceforge.net/ for the measurements.

```
hotspot=DetectHaarParam...()
host=genesis.cs.vt.edu:9999
mode=plain
```

```
hotspot=DetectHaarParam...()
host=genesis.cs.vt.edu:9999
mode=adaptive
criteria=energy
strategy=Compression
```

(a) Configuration files for the face recognition app.

```
hotspot=OCR.ImgOCRAndFilter()
host=genesis.cs.vt.edu:9999
mode=plain
```

```
hotspot=OCR.ImgOCRAndFilter()
host=genesis.cs.vt.edu:9999
mode=adaptive
criteria=energy
strategy=Compression
```

(b) Configuration files for the OCR app.

Fig. 11. Configuration files for the case study apps.

mobile network: 70ms RTT and 1Mbps bandwidth) [2]. Table II shows the device-specific values that parameterize the runtime systems of the mobile device under test.

TABLE II
MANUFACTURER PROVIDED ENERGY PROFILE.

CPU	1512.0 MHz: 577 mA	**WiFi**	96 mA
	1209.6 MHz: 408 mA		0.3 mA
	907.2 MHz: 249 mA	**Mobile**	250 mA
	604.8 MHz: 148 mA		3.4 mA
	302.4 MHz: 55 mA		

C. Third Party Applications

To determine if the implemented runtime can improve the energy efficiency of real-world mobile applications, we experimented with open source projects as our experimental subjects. JJIL[3] is a face recognition application; its recognition functionality executes remotely in class `DetectHaarParam`. Mezzofanti[4] is a text recognition application; its OCR functionality executes remotely in class `OCR`.

Furthermore, to achieve additional energy savings, we enhanced our runtime system with a module that compresses the transferred program state. This change did not require any other modifications to the implementation. Data compression can reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Transmitting raw data increases network transfer, but requires less CPU processing. Which of the strategies will consume less energy depends on the runtime conditions in place.

For each subject, we measured the amount of the energy consumed or the execution time by typical, simple use cases. Specifically, for the face recognition application, we examined one image file for the presence of human faces. For the OCR application, we examined one image file containing about 200 characters. The use cases were executed under three optimization modes: (1) original application, (2) plain cloud offloading, (3) adaptive cloud offloading. Figure 11 shows the configurations used in this case study, and Figure 12 shows code snippet of the compression strategy.

Figure 13 shows how the implemented runtime system helped reduce the amount of energy consumed by the face recognition application. In this experiment, the runtime system compresses the transferred state. The runtime system reduced the amount of energy consumed by ∼44% as compared to its local version and by ∼62% as compared to its plain offloading version, respectively. In the low-end mobile network, the plain cloud offloading optimization always consumes more energy than its local version due to the network transmission overhead. However, the amount of energy consumed by the adaptive cloud offloading optimization does not exceed its local version.

Figure 14-(a) shows how the implemented adaptive runtime system helped reduce the amount of energy consumed by the OCR application in both high- and low-end networks. Because in the high-end mobile network the compression strategy incurs additional processing overhead (i.e., 4.5 times

```
public class Compression extends Strategy {
  public Pointcut getPointcut() { return Pointcut.Around; }

  public Object invoke(Invocation invocation) {
    Object[] params = invocation.getParams();
    /** compress parameters */
    loop { //compress each parameter
      ByteArrayOutputStream baos = ... ;
      GZIPOutputStream gzipOut = ... ;
      ObjectOutputStream objectOut = ... ;

      objectOut.writeObject(params[i]);
      params[i] = baos.toByteArray();
    }
    point.setParams(params)

    /** Carry out the remote invocation, blocking for result */
    Object result = invocation.proceed(Proxy.MODE_BLOCK);

    return result ;
  }
}
```

Fig. 12. Compression strategy implementation.

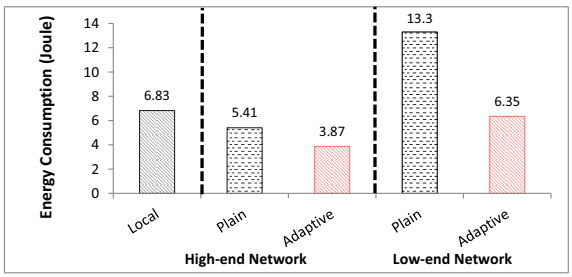
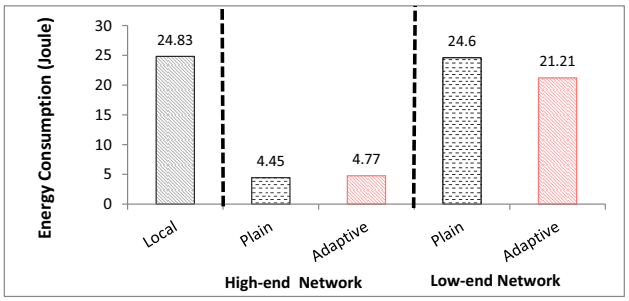


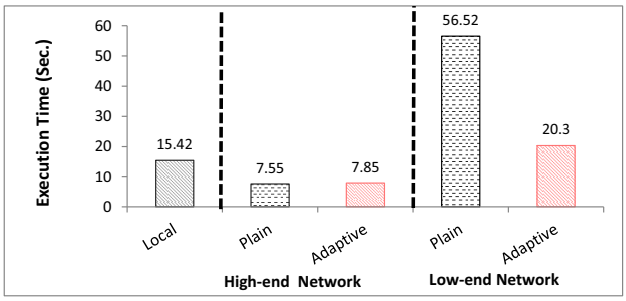
Fig. 13. Energy consumption of face recognition app on the high-end device.

[2]We used Network Emulator for Windows Toolkit (NEWT) version 2.1.
[3]http://code.google.com/p/jjil/
[4]https://code.google.com/p/mezzofanti

larger than with plain cloud offloading), the runtime system does not compress any state. In the other network condition, the runtime system reduced the amount of energy consumed by 15% through the compression strategy. However, as shown in Figure 14-(b), the both adaptive and plain versions could not reduce the total execution time taken by the OCR application in the low-end network as compared to its their local version. The performance overhead of a plain version is 266% while the adaptive version spent 31% more time. If an application is configured to favor performance over energy efficiency (e.g., `performance`), the runtime system will not initiate offloading operations in the presence of poor network conditions.



(a) Energy consumption.



(b) Execution time.

Fig. 14. Energy consumption and execution time of the OCR application.

VI. Discussion

Next, we discuss the approach's advantages and limitations.

A. Advantages

The presented approach to facilitating the implementation of adaptive cloud offloading optimizations provides multiple advantages, from both systems and software engineering perspectives. Among the specific benefits of using the described system architecture featuring reusable blocks accessible via an intuitive programming interface include expressiveness, generality, separation of concerns, and reusability. When implementing their own adaptive cloud offloading optimizations, the provided building blocks free the programmers to focus on the algorithmic and system design aspects of their implementation. This approach is general in that it can be applied to optimize the energy efficiency of a variety of existing mobile applications. Finally, this approach enables a greater separation of concerns in that it can change a mobile applications's energy/performance characteristics without affecting its core business logic.

B. Limitations

Although our approach also has some limitations. Because we estimate the energy consumption and performance efficiency parameters at the software level model, the estimation modules may not report the actual amounts of energy consumed by mobile applications. In addition, our energy model only takes CPU and network information, so that the energy consumption of a method containing significant file I/O or using sensors may turn inaccurate. Finally, while our system architecture provides fine-tuned reusable implementation blocks, the quality of end-product runtime system still remain highly dependent on the programmer's skills. Overall, we foresee that using our architecture effectively would still require a certain degree of competency and expertise from the programmer.

VII. Related Work

The work presented here is related to other complementary efforts to optimize the energy efficiency of mobile applications. Other related approaches have proposed programming models to facilitate the development of complex computer systems in different domains. Due to space limitations, we next compare and contrast our work with only representative samples of these complementary efforts.

A. Energy Optimization Mechanisms

Cloud offloading has recently enjoyed a lot of attention in the research literature with multiple competing approaches being advocated. CloneCloud [12] offloads execution at the thread level, while Cloudlet [13] offload at the VM level. ThinkAir [7] provides a tool chain to offload energy intensive methods to the cloud and scales up the resulting cloud-based execution by running the offloaded methods in parallel on dynamically allocated virtual machines. Recent systematic literature reviews such as [14] provide insights on the strengths and weaknesses of major offloading mechanisms, including CloneCloud, Cloudlet, and ThinkAir. System designers can leverage these insights when they use our reusable building blocks to construct a runtime system for cloud offloading that meets their specific requirements. The goal of this work is to facilitate these various cloud offloading implementations by providing reusable building blocks.

A fuzzy decision model has been proposed as an alternate mechanism for steering cloud offloading adaptations [15]. Our system architecture can flexibly support a variety of decision models including a fuzzy one. In a way, this work is complementary to existing state-of-the-art offloading mechanisms in its focus on providing reusable software building blocks that are agnostic of the offloading adaptation mechanism in place.

In addition to system-level solutions, programming-level solutions (e.g., energy saving algorithms [16], design patterns for energy efficient computing [17], software models for energy efficient software [18], programming languages for energy

efficiency [19]) have also been advocated in the literature. We see our approach as lying on the intersection of system- and program-level solutions. Our system architecture and programming model enable the creation of powerful system-level cloud offloading optimization by providing convenient software abstractions exposed as a library.

B. Programming Models for Systems

Our approach follows a long tradition of creating system architectures and programming models to facilitate the construction of complex system solutions. In distributed systems, component-based programming frameworks have been used successfully as a means of reducing the complexity of constructing large-scale heterogeneous systems (e.g., CORBA [20], enterprise messaging [21], Web services [22], etc.).

In addition to these general models for distributed systems, specialized models have been proposed to express complex distributed system functionalities. FarGo-DA [23] is a framework that provides components with disconnection and reconnection semantics to operate mobile applications in the presence of network disconnection. To address the complexity, heterogeneity and dynamism of grid environments, Accord [24] provides autonomic components to enable programmers to cleanly separate the concerns of grid management from that of core computation. Mobile Fog [25] provides programming abstractions for future Internet applications; it utilizes cloud computing mechanisms to enable runtime scalability, thus enabling programmers to construct applications that are distributed, large-scale, and latency sensitive.

Similarly to these approaches, our overriding goal is to help programmers optimize the execution of their mobile applications. To that end, we created a system architecture with intuitive programming abstractions. A key difference of our approach as compared to the prior state of the art is our focus on improving energy efficiency. Although energy efficiency has become an intrinsic consideration of mobile system design, energy-efficiency remains relatively unexplored from the software architecture and programming support perspectives. Hence, this research aims at addressing this deficiency, thus improving the overall quality of mobile software.

VIII. Conclusion

This research focuses on the problem of engineering adaptive cloud offloading optimizations to improve the energy efficiency of mobile applications. Although cloud offloading has been shown as an effective energy optimization technique, its engineering remains a complex system construction undertaking. To address this problem, we presented a system architecture and a programming model that provide reusable building blocks that can streamline the process of constructing cloud offloading optimizations. The results of our case study and energy efficiency evaluation indicate that our system architecture and programming model make it possible to implement highly effective cloud offloading optimizations. By facilitating the process of implementing cloud offloading optimizations, we hope to be able to add the cloud offloading optimization in the standard toolset of mobile application programmers.

Acknowledgments

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

References

[1] Facebook Mobile, "Facebook for every phone," July 2011.

[2] Y.-W. Kwon and E. Tilevich, "The impact of distributed programming abstractions on application energy consumption," *Information and Software Technology*, vol. 55, no. 9, pp. 1602–1613, 2013.

[3] K. Yang, S. Ou, and H.-H. Chen, "On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications," *Communications Magazine, IEEE*, vol. 46, no. 1, pp. 56–63, 2008.

[4] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Enhancing Java programs with distribution capabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 1, pp. 1–40, 2009.

[5] R. Johnson, R. Helm, J. Vlissides, and E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[6] Y.-W. Kwon and E. Tilevich, "Reducing the energy consumption of mobile applications behind the scenes," in *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, 2013.

[7] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of IEEE INFOCOM*, 2012.

[8] Y.-W. Kwon and E. Tilevich, "Energy-efficient and fault-tolerant distributed mobile execution," in *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[9] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 48–63, 1999.

[10] 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*, vol. 25, no. 4, pp. 722 –737, 2007.

[11] E. Tilevich and Y. Smaragdakis, "NRMI: Natural and efficient middleware," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 2, pp. 174–187, 2008.

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

[13] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[14] G. A. Lewis, P. Lago, and G. Procaccianti, "Architecture strategies for cyber-foraging: Preliminary results from a systematic literature review," in *Proceedings of the 8th European Conference on Software Architecture (ECSA 2014)*, 2014, pp. 154–169.

[15] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceeding of the 4th ACM Workshop on Mobile Cloud Computing and Services (MCS 2013)*, 2013, pp. 9–16.

[16] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "A preliminary study of the impact of software engineering on greenIT," in *Proceedings of the 1st International Workshop on Green and Sustainable Software*, 2012.

[17] Y. Liu, "Energy-efficient synchronization through program patterns," in *Proceedings of the 1st International Workshop on Green and Sustainable Software*, 2012.

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

[19] 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.

[20] Object Management Group, "The CORBA component model specification," Object Management Group, Specification, 2006.

[21] R. Monson-Haefel and D. Chappell, *Java Message Service*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000.

[22] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web services*. Springer, 2004.

[23] Y. Weinsberg and I. Ben-Shaul, "A programming model and system support for disconnected-aware applications on resource-constrained devices," in *Proceedings of the 24rd International Conference on Software Engineering (ICSE)*, 2002.

[24] H. Liu, M. Parashar, and S. Hariri, "A component-based programming model for autonomic applications," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2004.

[25] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing*, ser. MCC '13, 2013.