Constraint-Driven Dynamic Adaptation of Mobile Applications for Quality of Service

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. Modern mobile applications are executed in a variety of execution environments by users with different preferences for energy savings, performance efficiency, reliability, and privacy. Offloading a mobile application's functionality to execute at a remote server has become an important energy and performance optimization technique. Mobile applications, however, executed over networks with divergent latency/bandwidth characteristics, access cloud-based servers that offer different levels of performance, availability, and privacy. An effective offloading mechanism must consider all these factors when determining which functionality should be offloaded to which server. In addition, offloading can be used to save energy, improve performance, or both. Implementing an adaptive offloading mechanism driven by both runtime conditions and user preferences is non-trivial. In this paper, we present a novel approach to configurable, adaptive offloading for mobile applications that is driven by constraint solving. The programmer annotates energy intensive functionality at the method boundary. The end user, via a configuration menu, specifies how to prioritize energy savings, performance efficiency, server availability, and privacy. The specified priorities are then automatically translated into constraints used at runtime by a third-party constraint solver to drive an adaptive offloading runtime system.

Key words: mobile applications, cloud offloading, energy optimization, constraint-solving, adaptation, software reengineering

1 Introduction

Mobile computing is characterized by a high heterogeneity of the hardware/software stack and network environments. That is, the same mobile application can be executed on mobile devices with different hardware configurations, varying in terms of their respective CPU, memory, and communication characteristics [8]. Another source of heterogeneity are mobile operating systems, whose implementation policies can affect the performance/energy profiles of the executed applications [16]. Finally, mobile devices use a great variety of network types with divergent characteristics, including latency, bandwidth, congestion, packet loss, and interference. Another prominent trend of mobile computing is the appli-

cations' energy demands outpacing the devices' battery capacities. Rapid growth in application functionality requires ever greater energy budgets, thus subsuming any advances in battery capacities.

A common energy optimization technique for mobile applications is cloud offloading—placing energy-intensive functionality to run at a remote, cloud-based server. Executing this functionality at a remote server saves the mobile device's battery power. The superior computational power of offloading servers also typically increases the performance efficiency of the offloaded functionality. As a result, cloud offloading has become a versatile optimization tool; programmers can use it to minimize energy consumption, maximize performance, or maximize some given energy-performance ratio.

By taking advantage of distributed execution, cloud offloaded applications can become subject to partial failure and privacy vulnerabilities. In particular, mobile networks are prone to volatility, with fluctuations in latency, bandwidth, and congestion. Depending on their location and administrative procedures, offloading servers may offer different levels of trust. Thus, while providing energy optimization advantages, cloud offloading also incurs problems traditionally associated with distributed execution.

These realities of mobile execution give rise to two fundamental problems of engineering effective cloud offloading optimizations: (1) how can the application programmer express the desired optimization priorities? (2) how can the system programmer create a runtime adaptation mechanism to drive offloading optimizations that takes into account both the expressed optimization priorities and the runtime environment in place? Solving these problems requires overcoming the high level of complexity imposed by the need to consider multiple parameters, both static and dynamic. These parameters need to be efficiently obtained and evaluated. Furthermore, the considered parameters may change depending on users, devices, and deployments. In the end, the goal of adaptive cloud offloading is to decide which portion of the mobile application's functionality to offload to which server.

In this paper, we present a novel dynamic adaptation approach for cloud offloading that leverages constraints satisfaction. We express the optimization priorities of cloud offloading as *a constraint satisfaction problem (CSP)*. Informally, a CSP computes the values that a set of variables must take in order to satisfy a set of conditions imposed on the variables. Our system architecture maps variables to offloading optimization criteria (e.g., energy savings, performance efficiency, server availability, server trustworthiness, etc.); values to the actual runtime parameters of the criteria (e.g., the amount energy consumed by a method, the time taken to execute a method, the average failure rate for an offload server, and the user-specified degree of trust for a server); conditions to the end user's specified optimization priorities (e.g., minimize energy consumption, minimize execution time, maximize a given energy/performance ratio, prefer offload servers with higher trust levels).

This paper makes three technical contributions: (1) a software model for constraint-solving driven adaptive cloud offloading, (2) a reference implementa-

tion of adaptive runtime system based on our model, and (3) empirical evaluation through a series of micro-benchmarks and case studies of optimizing the energy efficiency of third-party mobile applications.

The rest of this paper is structured as follows. Section 2 presents a real world scenario applying our approach and introduces the concepts and technologies used in this work. Section 3 details our technical approach. Section 4 presents the results of evaluating our approach. Section 5 discusses the advantages and limitations of our approach. Section 6 compares our approach to the related state of the art, and Section 7 presents concluding remarks.

2 Real World Scenario and Technical Background

In this section, we first present a real world scenario that motivates our approach. Then, we introduce the main technical concepts used in this work.

2.1 Real World Scenario

Consider traveling to a foreign country whose language you cannot read. Not being able to read public signs in an unfamiliar language (e.g., Chinese for an English-speaking visitor) is likely to hinder one's traveling experiences. Mezzofanti is an Android application that solves this problem; it translates signs taken as a picture with the mobile device's camera. The taken pictures parameterize an optical character recognition (OCR) algorithm that extracts the text contained in the picture. The application then translates and presents the extracted text using automatic language translation. Because of the energy-intensive OCR functionality, using the Mezzofanti application heavily can quickly drain the device's battery.

As consuming an inordinate amount of energy, the OCR functionality makes a promising candidate for cloud offloading. That is, instead of running the OCR algorithm on the mobile device, text images can be transferred across the network to the cloud server, so that the OCR functionality would be executed remotely, without draining the device's battery power. The extracted text can then be transferred back to the device to be automatically translated. Because the server processing capacities (i.e., CPU speed, memory size, cache architecture, etc.) are more powerful than those on the device, the offloaded OCR functionality is also bound to improve its performance efficiency.

Nevertheless, one must ask whether the OCR functionality should be offloaded once and for all. For example, when a mobile device is disconnected, offloading optimizations render applications unusable. When the network connection is poor, the energy savings afforded by the offloading optimization will quickly disappear, as retransmitting lost packets over a volatile cellular network can tap deeply into the application's energy budget. By contrast, in the presence of a stable network connection, an offloading optimization can be used to reduce energy consumption, to maximize performance, or to pursue an optimization strategy that correlates both of these objectives.

Consider the following three scenarios of using Mezzofanti that would prioritize the offloading of the OCR functionality in different ways:

1. When using the application within a close proximity to the hotel, the user may want to *maximize both energy savings and performance efficiency*. Running out of battery power would not be catastrophic, as the user can come back to the hotel and recharge the device.
2. When riding a city bus and using the application to make sense of public service announcements, the user may want to *maximize performance efficiency*. If an announcement, for example, conveys the intent to alter the original service route, the user needs to learn this information as soon as possible to have enough time to get off at the very next stop if necessary.
3. When exploring the areas far away from the hotel, the user may want to *maximize energy savings*. The user needs to keep the device operational for as long as it takes to get back to the hotel.

Our work addresses the need for scenario-specific configurability for adaptive cloud offloading optimizations as demonstrated by the scenarios above.

2.2 Technical Background

The major technologies used in this work are cloud offloading and constraints satisfaction. We describe them in turn next.

Cloud Offloading Cloud offloading has become a popular optimization technique for mobile applications. It leverages the resources of cloud-based remote servers to execute portions of a mobile application's functionality. By executing some of the application's functionality in the cloud, offloading reduces the amount of energy consumed by the mobile device, thus saving its battery power. An additional benefit of cloud offloading is improved performance efficiency, as cloud servers have hardware resource more powerful that those available on mobile devices. Application-level cloud offloading optimizations are typically implemented by partitioning the application into the client and server parts communicating across the network.

In our prior work in cloud offloading, we introduced a novel partitioning mechanism that leverages static program analysis and program transformation techniques to optimize a transferred program state [7]. In addition, we introduced an adaptive offloading mechanism, in which the local/remote application parts are determined at runtime, as driven by the execution environment in place [9].

Constraint Satisfaction Problem Constraint satisfaction problem (CSP) is a mathematical model that describes optimization scenarios involving objects and their states; various constraints are imposed on the states, and the solution must satisfy all the constraints. Formally, a CSP involves finite sets of domains, variables, and constraints. A domain defines a finite set of values or value ranges. Each variable is assigned to a domain. A constraint is a predicate expressed in first-order logic. A constraint expresses the condition of assigning a domain to

variables. To solve a CSP is to find an assignment that satisfies all the specified constraints. A common variant of CSP is the boolean satisfiability problem (SAT), which uses a boolean formula to solve a CSP.

As a specific example, consider a domain $D = \{1, 2, 3\}$, assigned to variables X and Y (i.e., $X \in D$ and $Y \in D$). We want to satisfy the following two constraints: $C_1 : X \neq Y$ and $C_2 : X < Y$. The first solution for this CSP is $X = 1$ and $Y = 2$. To find the second solution (i.e., $X = 2$ and $Y = 3$), we would have to add another constraint $C_3 : (X \neq 1) \wedge (Y \neq 2)$. In terms of its time complexity, a general CSP is NP-complete. However, practical heuristics have been created to solve CSPs in a reasonable time.

3 Constraints-Driven Adaptive Cloud Offloading

In this section, we present constraints-driven adaptive cloud offloading, a new approach to enhancing mobile applications with adaptive energy optimization capabilities. Then, Section 4 presents the empirical results of applying our approach to third-party mobile applications.

3.1 Approach Overview

Our approach enhances third-party applications with the ability to optimize their execution via user-configured adaptive cloud offloading. Figure 1 outlines the design space for this work, whose key objective is to flexibly adapt offloading optimizations for the inherent variabilities of distributed mobile execution. Specifically, cloud offloading environments are characterized by the presence of heterogeneous mobile devices, offloading servers, mobile/cellular networks, and user privacy preferences.

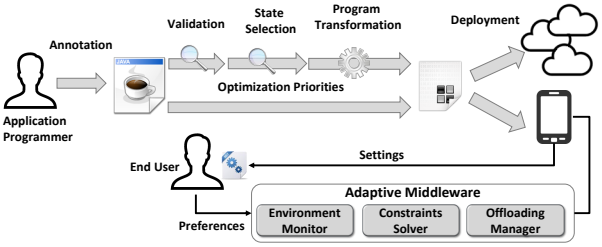


Fig. 1. Overall procedure for the proposed approach.

Figure 1 shows our approach's main workflow. The runtime adaptivity of the offloading behavior is parameterized by both application programmers and end users. The application programmer's responsibility is to identify the methods that impose a high energy overhead and would make promising offloading candidates. How the programmers identify these methods is orthogonal to our approach—they can take advantage of existing energy profiling tools [17, 10]. The identified methods are marked by annotation `@OffloadingCandidate`. Based on their application knowledge, programmers provide additional configuration parameters using the `Selection` annotation, which defines the criteria of `EnergySavings`—minimize the amount of energy consumed, `PerformanceEfficiency`—minimize the execution time, `Availability`—favor the offloading servers that are

more likely to be available, `Privacy`—favor the servers deemed as more trustworthy. The following code snippet presents an example of annotating a method:

```
public class OCR {
  @OffloadingCandidate
  @Selection(criteria={EnergySavings|PerformanceEfficiency|Availability})
  public void ImgOCRAndFilter(...) { ... } }
```

The programmer identified this method as an energy hotspot suitable for a cloud offloading optimization. The programmer also specified that the offloading should be parameterized with the optimization criteria of energy savings, performance efficiency, and server availability. These criteria are then made available to end users, who determine the actual runtime behavior of the identified offloading optimization.

The annotated application is then analyzed and transformed by going through the following steps. The programmer specified annotations are first verified to make sure that the identified candidate methods can be safely offloaded (i.e., they do not rely on device-specific resources such as sensors). Then a static program analysis determines the program state that must be transferred to the server and back. Based on the analysis' results, a bytecode enhancer modifies the compiled application to generate the necessary checkpoints[1].

Based on the offloading criteria specified by the `@Selection` annotation, a new settings menu is added to the optimized application. Once the end user selects the optimization criteria, a generator, also added to the application, synthesizes a set of constraint declarations for each specified criteria. The details of generating the constraint declarations appear in Section 3.2.

The enhanced application is installed on the mobile device and on each available offloading server. A simple configuration file installed on the mobile device lists the URLs of the available offloading servers, their respective trustworthiness rankings (in percents), and the availability threshold (i.e., the availability rate bellow which the server should not be considered). This file should be provided by a person knowledgeable about the application's execution environment, such as a system system administrator or a sophisticated end user. In addition to the enhanced application, our approach also distributes a small runtime system parameterized with the synthesized constraint specifications. The runtime system includes an environment monitor to keep track of the execution environment, a constraint solver to determine offloading strategies at runtime, and an offloading manager to handle network communication and state synchronization. The details of the runtime system appear in Section 3.3.

3.2 Generating and Evaluating CSP Constraint

Recall that CSP constraints are boolean predicates that can be efficiently evaluated by a constraint solver. Our approach uses constraints to express commonly accepted invariants of the cloud offloading optimization. For example, to save

[1] We employed the same static program analysis and program transformation mechanisms developed through our prior work [7], [9].

energy by offloading a method to the cloud, the amount of energy consumed by the network transfer must be lower than the amount of energy consumed by executing the method locally. Similarly, to increase performance by offloading a method to the cloud, the time taken by the network transfer must be less than the time taken by executing the method locally. With respect to server availability, the server with the highest observed availability rate should be selected. With respect to server privacy, the server with the highest level of trustworthiness, as specified by the system administrator, should be selected. The reference implementation of our approach provides the invariants for energy savings, performance efficiency, availability, and privacy.

The constraint definitions are generated at configuration time, once the end user has selected the optimization criteria. These constraint definitions parameterize a third-party constraint solver, whose role is to efficiently determine the offloading strategy that meets the user's preferences for a given distributed execution environment. Therefore, the constraints adhere to the common CSP format, which is accepted by many third-party constraint solvers. Even though the reference implementation uses the Sugar constraint solver [12], this system component is plug-in replaceable.

The user interface for expressing the optimization criteria enables the end user to specify an ordered set of either single criteria or correlations of criteria. The items appearing earlier in the set have higher priority. Consider the following two examples of user-specified optimization settings:

Example #1: The end user selects "Energy Savings" as the first criterion and "Performance Efficiency" as the second criterion. This selection forms an ordered set of two items and is interpreted as follows: (1) select offloading servers, executing a method on which would consume less energy than executing the method on the mobile device (i.e., by considering the network communication costs), (2) select offloading servers, executing a method on which would take less time than executing the method on the mobile device, (3) compute the intersection of the results from steps (1) and (2) (i.e., satisfying both criteria), and (4) select the highest ranking member of the intersection by favoring the first criterion over the second one.

Example #2: The end user selects "Energy Savings & Performance Efficiency" as the first criterion and "Availability" as the second criterion. This selection forms an ordered set of two items and is interpreted as follows: (1) select offloading servers, executing a method on which would yield higher energy/performance ratio than executing the method on the mobile device, (2) select offloading servers whose observed availability rates are higher than the system administrator-specified availability threshold, (3) compute the intersection of the results from steps (1) and (2), and (4) select the highest ranking member of the intersection by favoring the first criterion over the second one.

Example #3: The end user selects "Performance Efficiency" as the first criterion, "Energy Savings" as the second criterion, and "Availability" as the third criterion. This selection forms an ordered set of three items and is interpreted as follows: (1) select offloading servers, executing a method on which would take

less time than executing the method on the mobile device, (2) select offloading servers, executing a method on which would consume less energy than executing the method on the mobile device, (3) select offloading servers whose observed availability rates are higher than the system administrator-specified availability threshold, (4) compute the intersection of the results from steps (1), (2) and (3) (i.e., satisfying all three criteria), and (5) select the intersection's highest ranking member favoring the first criterion over the second one, and the second over third.

To see how this selection process works with specific runtime parameters, consider applying the following runtime environment to the two examples above. Table 1 presents three runtime parameters: energy consumption, execution time, and availability rate for **Local** execution on the mobile device, and on three offloading servers, **S1**, **S2**, **S3**. The parameters are dynamically estimated by our adaptive runtime system, whose design is detailed in Section 3.3.

Table 1. Example runtime environment.

	Local	**S1**	**S2**	**S3**
Energy Consumption (Joule)	20	15	17	22
Execution Time (ms)	250	200	125	100
Availability Rate (%)	100	90	85	98
Energy/Performance Ratio	30	50	88	54

In **Example #1**: (1) The energy savings constraint will select servers **S1** and **S2**, whose energy consumption numbers of 15 and 17 are smaller than that of **Local** (i.e., to be executed on the mobile device). (2) The performance efficiency constraint will select servers **S1**, **S2**, and **S3**, whose estimated execution time is shorter than that of **Local**. (3) The intersection between these results is **S1** and **S2**. (4) Select **S1**, as its energy consumption is lower than that of **S2**.

In **Example #2**: (1) The energy and performance constraint will select servers **S1**, **S2** and **S3**, whose respective energy/performance ratios are higher than that of **Local**. (2) The availability constraint will select servers **S1** and **S3**, whose observed availability rates are higher than the 90% user-specified threshold. (3) The intersection between these results is **S1** and **S3**. (4) Select **S3** over **S1** as having a higher energy/performance ratio.

In **Example #3**: (1) The performance efficiency constraint will select servers **S1**, **S2**, and **S3**, whose estimated execution time is shorter than that of **Local**. (2) The energy savings constraint will select servers **S1** and **S2**, whose respective energy consumption numbers are smaller than that of **Local**. (3) The availability constraint will select server **S3**, whose availability rate is higher than the specified threshold of 95%. (4) The intersection between these results is empty. (5) Execute locally (however, if the availability rate of **S2** were to increase to above 95%, it would be selected).

Even though constraint solving is known to be an NP-Complete problem, the limited number of constraints used in a typical cloud offloading scenario makes the approach not just feasible but quite efficient. Running the constraint solver incurs energy and performance overheads not surpassing more than a couple

of percentage points of the overall energy/performance budgets. We detail our assessment of the constraint solving overheads in Section 4.

Generating constraint definitions, a one-time expense incurred at the configuration time, is linear in complexity, in which a predefined template is just filled in with application-specific parameters. Figure 2 shows a specific example of generating a constraint definition from a template.

```
; BEGIN #name
; nbDomains=#num
#domains
; nbVariables=#num
#variables
; nbPredicates=#num
#predicates
; nbConstraints=#num
#constraints
; END #name
```

```
; BEGIN EnergyAndPerformance
; nbDomains=2
(domain D1 (#energy))
(domain D2 (#performance))
; nbVariables=2
(int V1 D1)
(int V2 D2)
; nbPredicates=2
(predicate (P1 X) (le X #e_local))
(predicate (P2 X) (le X #p_local))
; nbConstraints=2
(P1 V1) ; C1
(P2 V2) ; C2
; END EnergyAndPerformance
```

Fig. 2. CSP template and constraint files.

3.3 Adaptive Runtime System

Our approach hinges on the ability to adaptively offloading functionality at runtime based on the current environmental conditions. This ability is provided by an adaptive runtime system that we describe next. The runtime system runs on the devices and provides monitoring, estimation, and offloading services. It monitors network delay, network connection type, CPU frequency, CPU time, and voltage. It estimates the expected energy consumption and execution time of running the offloaded method on each offloading server. It offloads methods, transferring and synchronizing the required program state. Running on the mobile device, the runtime system must exhibit low energy and performance overheads to be practical.

Major Components Figure 3 shows a component diagram of the runtime system. The functionality is encapsulated within five modules: monitoring, constraint solving, state management, network, and execution history.

The monitoring module is responsible for monitoring the execution environment, both of the mobile device and of the network. The system monitoring unit taps into the platform diagnostics API

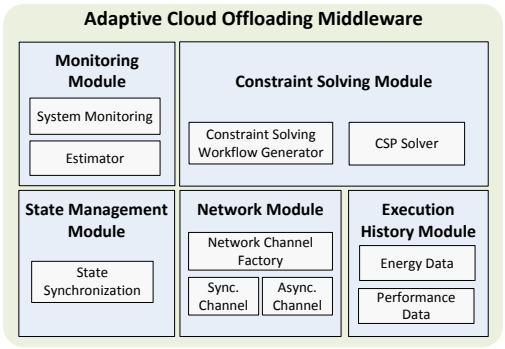


Fig. 3. Adaptive runtime system.

to periodically obtain the values of network connection type and current voltage. The CPU usage time and frequency are retrieved from `proc\[pid]\stat`. The current network delay as well as the execution and idle times of network communication are measured by sending probe packets to remote servers.

To predict the amount of energy to be consumed during an offloading, the estimator correlates previously measured energy consumption values and the currently measured value:

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

where E_{prtd} is the estimated future energy consumption, E_{cpu}^{avg} is the average energy consumption value of the offloading operation, $T_{net}^{est\_act}$ is the estimated communication time, and $T_{net}^{est\_idle}$ is the estimated idle time, respectively. V is the voltage reported by platform-specific battery APIs (e.g., the `BatteryStat` class on Android). Finally, the estimator computes the expected execution time by averaging the prior execution time and predicting the communication time.

Having completed an offloading operation, the system monitoring unit measures the amount of energy consumed:

$$\boldsymbol{E} = \{\Sigma(C_{cpu@f}^{act} \times T_{cpu}^{(u+s)}) + (C_{net}^{act} \times T_{net}^{act}) + (C_{net}^{idle} \times T_{net}^{idle})\} \times V$$

where $C_{cpu@f}^{act}$ is the electric current of the CPU at a given clock speed. T_{cpu}^{u} and T_{cpu}^{s} are the user and system times of an application process, respectively. C_{net}^{act} and C_{net}^{idle} are the electric current of the network processor needed during the active and idle phases, respectively. T_{net}^{act} and T_{net}^{idle} are the measured active and idle times during the offloading operation, respectively. The energy consumption and performance are continuously measured and cached for use in subsequent estimations, thus improving the accuracy of the estimation process.

The constraint solving module is responsible for running the constraint-solving workflow, which computes solutions that satisfy the current constraints. The constraint workflow generator parameterizes the generated constraint definitions with the actual runtime values obtained from the monitoring module, so that the CSP solver always works with the most up-to-date runtime information.

The state management module is responsible for synchronizing the program state. The program state is synchronized by using `copy-restore`, a parameter passing semantics for remote methods that is applicable to

```
delays ← checkDelays(URLs);
FOREACH S_n ∈ ∀S DO
  D_url ← delays.getDelay(url)
  E_prtd ← computeEnergyConsumption(D_url, S_n)
  T_prtd ← computeExecutionTime(D_url, S_n)
  updateConstraints(url, E_prtc, T_ptrd)
END FOREACH

/** Determine a server & an offloading unit */
offloadingServer, method ← solveConstraints()

/** Store the current program state */
toServer ← checkpointCurrentState()

/** Send prog. state to the remote server */
sendToServer(offloadingServer, toServer)

/** Receive a new state or exception and
    notify of it to all relevant parties */
CASE Succeed
  fromServer ← offloadingCompleted(...)
  E_cnsmd, T_exec ←— endMeasurement()
  update(offloadingServer, toServer,
    fromServer, delays, E_cnsmd, T_exec)
  synchronize(fromServer, toServer)
CASE Fail
  exception ← offloadingFailed(...)
  update(url, toServer, exception)
```

Fig. 4. Adaptive cloud offloading operation.

complex linked data structures [15]. This semantics first copies the reachable program state to the server, and then efficiently synchronizes the client's state with the server modified data, while preserving all the client-side aliases.

The network module is responsible for managing connections between the client and the offloading servers. The network channel factory creates multiple network channels for each server. A network channel reports the measured network delay, sends/receives messages, and computes the time of each communication phase (i.e., sending, idle, and receiving time).

The execution history module is responsible for managing the offloading history data. It maintains the energy and performance caches, which are consulted by the estimator unit when computing the expected energy consumption and execution time numbers. The pseudo code in Figure 4 shows the runtime execution logic and the interactions between the modules described above.

4 Evaluation

This section describes the micro benchmarks and case studies that evaluate the effectiveness of our approach. The experimental setup includes a low-end mobile device (600MHz CPU, 256MB RAM, 802.11g), a high-end mobile device (1.5GHz dual-core CPU, 2GB RAM, 802.11n), and an offload server (3.0GHz quad-core CPU, 8GB RAM). Table 2 shows the device-specific values that parameterize the runtime systems of the mobile devices under test. To measure energy consumption, we used our energy model described in Section 3.3.

Table 2. 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			

4.1 Micro Benchmarks

Benchmark I: Runtime System Overheads In this benchmark, we executed empty remote methods passing to them three different payloads (100kB, 1MB, and 5MB) over the network (20ms latency and 50Mbps bandwidth), with and without the adaptive runtime functionality enabled, thus isolating the runtime system's performance and energy overheads [2]. Table 3 shows the results for each device. As expected, both the performance and energy overheads of the runtime system are proportional to the offloaded methods' payload (i.e., the size of the program state transferred). Since the average size of the program state transferred during an offloading is 2-3 MB, the corresponding energy/performance

[2] To emulate network conditions, we used Network Emulator for Windows Toolkit (NEWT) version 2.1.

overheads of ˜100mJ/˜300ms indicate one could use adaptive runtime system to drive offloading in practical settings.

Table 3. Performance and energy consumption overhead.

Payload		High-end device		Low-end device	
		Plain	Adaptation	Plain	Adaptation
100kB	**Energy**	16mJ	22mJ	41mJ	46mJ
	Time	160 ms	167ms	227ms	264ms
1MB	**Energy**	236mJ	214mJ	362mJ	407mJ
	Time	1132ms	1156ms	1355ms	1278ms
5MB	**Energy**	1643mJ	1828mJ	3012mJ	3287mJ
	Time	3371ms	3904ms	5192ms	5673ms

Benchmark II: Performance and Energy Consumption of Solving Constraints In this benchmark, we assessed the scalability of the constraint solver w.r.t. the number of constraints. Figure 5 shows that the solver's performance time and energy consumption grow linearly with the number of constraints. Since the number of constraints is not expected to exceed five in a typical application, the solver does not appear to be either an energy or performance bottleneck.

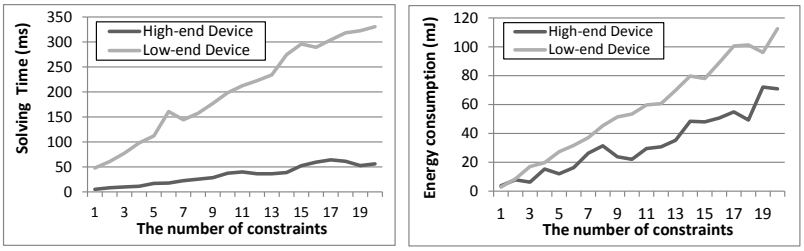
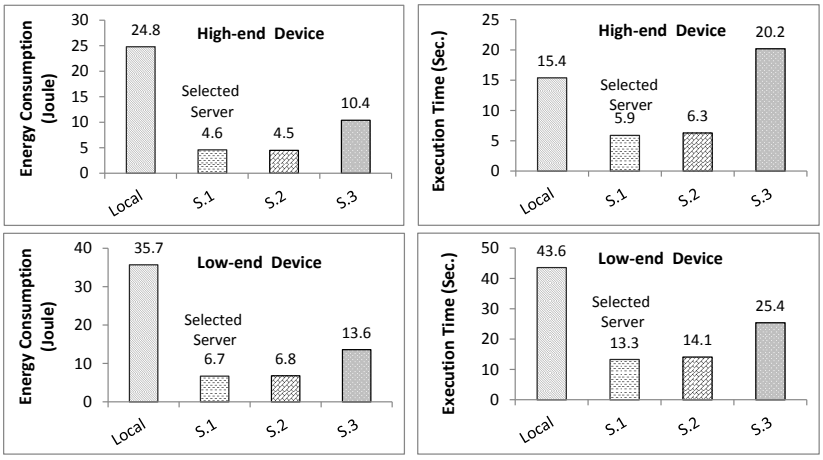


Fig. 5. Performance and energy consumption of the constraint solver.

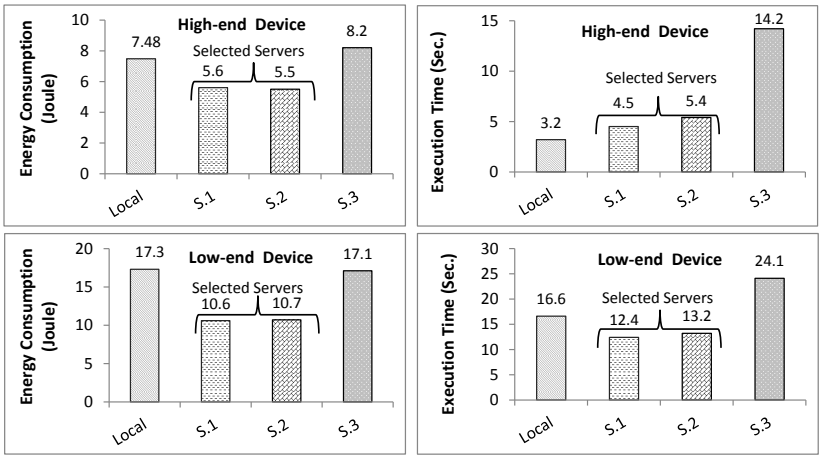
Benchmark III: Multi-Server Environment In this benchmark, we executed empty remote methods passing to them the 1MB payload over two types of networks with the following round trip time (RTT)/bandwidth ratios (Network I: 20ms/50Mbps; Network II: 50ms/3Mbps). While executing these methods, the runtime system ran its estimation algorithm taking into account an increasing number of offloading servers. As it turned out, the number of offloading servers does not significantly affect their respective runtime/energy efficiency due to the asynchronous architecture.

Table 4. Performance comparison when connecting multiple servers.

Server #	1	2	3	4	5
Network I	904ms	1064ms	978ms	1036ms	1154ms
Network II	1967ms	1960ms	1952ms	1991ms	2111ms



(a) Energy consumption and execution time of the OCR app.



(b) Energy consumption and execution time of the face recognition app.

Fig. 6. Experimental results of the subject applications.

4.2 Case Study

To determine if our approach can improve the energy efficiency of real-world mobile applications, we experimented with open source projects as our experimental subjects. Mezzofanti[3], described in Section 2, extracts and translates text from images; we marked its method `ImcOCRAndFilter` in class `OCR` as `@OffloadingCandidate` with the `@Selection` criteria set to both `EnergySavings` and `PerformanceEfficiency`. JJIL[4] recognizes faces from images; we marked its method `push` in class `DetectHaarParam` as `@OffloadingCandidate` with the `@Selection` criteria set to both `EnergySavings` and `PerformanceEfficiency`. A separate, preceding profiling procedure determined the annotated methods as energy and computation intensive.

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

Table 5. Emulated execution environment settings.

	S1	S2	S3
Avg. latency(ms)/bandwidth(Mbps)	5/50	5/50	20/3
Additional exec. time (ms)	0	1000	0

Figure 6 shows how our approach has reduced the amount of energy consumed by the subjects. The amount of energy consumed and the execution time were measured. For each subject, we present four graphs showing the amount of the energy consumed by their canonical use cases. Specifically, the OCR application examines one text image file containing about 200 characters. The face recognition application examines one image file for the presence of human faces. The experiment assumes that the end user has configured the OCR application for `energy+performance` and the face recognition application for `energy savings`. Table 5 summarizes the emulated experimental environments for each of the three offloading servers.

In this case study, we assessed whether our adaptive offloading mechanism would select the most appropriate offloading server to satisfy the specified user preferences. To that end, we modified the runtime systems' implementation to always offload the annotated to all the available servers. This way, we could measure the actual energy consumption and execution time for each offloading scenario. We also recorded which offloading server was selected by the constraint solving module. Thus, we evaluated the effectiveness of our offloading selection mechanism in the presence of complete knowledge about the resulting energy/performance gains provided by each server. For the OCR application (Figure 6 (a)), our constraint-solving based offloading selection mechanism always chose the server that maximized the optimization criteria in place. For the face recognition application (Figure 6 (b)), our selection mechanism chose either the best or the second best offloading option. This variability stems from optimizing only for energy savings, with the actual energy consumption levels being quite close for the two top options (within 3%).

For the next experiment, we studied the impact of changes in network conditions on the energy consumption and execution time improvements afforded by offloading optimizations. To that end, we compared the respective effectiveness of the plain and adaptive offloading mechanisms. The user preferences for the face recognition application were set to `energy+performance`. The main application loop was executed 10 times, divided equally into two phases. The network

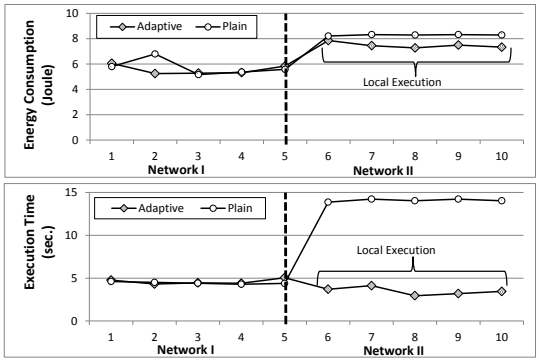


Fig. 7. Experimental results of the face recognition app. when changing network conditions (i.e., 20ms/50Mbps → 50ms/3Mbps).

conditions (delay/bandwidth) were emulated for the first phase as 20ms/50Mbps and for the second one as 50ms/3Mbps. As shown in Figure 7, during the first phase (favorable network conditions), both plain and adaptive offloading schemes were equally effective. However, during the second phase (poor network conditions), the adaptive scheme turned more effective, particularly for performance. Indeed, in the presence of a poor network condition, executing locally, without any offloading, turned to be the optimal strategy.

5 Discussion

How one can dynamically adapt mobile applications for execution environments and users is a hard problem, and we do not claim that constraint solving is *the solution*. By discussing our approach's advantages and limitations, we strive to highlight the complexity of the target domain and the challenges to be overcome.

5.1 Advantages

As compared to the related state of the art on cloud offloading, our approach offers a high degree of configurability. The adaptive behavior is configured first by the application programmer and then by the end user. Furthermore, complex adaptivity requirements are expressed via intuitive interfaces. Specifically, application programmers annotate energy and performance intensive methods, while end users use a GUI-based settings dialog automatically added to the application based on the annotations.

The mechanism that translates these declarative specifications into sophisticated runtime behavior is our adaptive runtime system. The use of a third-party constraint solver streamlines the process of evaluating multiple complex conditions, making it robust and efficient. The runtime system implementation imposes a low energy and performance overhead on the underlying application by relying on system-provided facilities for querying the runtime information.

5.2 Limitations

By relying on CSP, our approach has limited scalability, as CSP is an NP-Complete problem in general. When in an experiment, we increased the number of constraints to 20, the solver's performance on a high-end mobile device remained practical. However, in our broader evaluation, with the number of constraints not exceeding 4, the solver's performance was never an issue.

Another limitation stemming from our use of CSP is that our approach cannot express degrees and ranges as constraints. Because we use an SAT-based constraint solver, all our constraints are `boolean` predicates. However, using `boolean` predicates turned to be quite suitable for mobile devices that are known for their heterogeneous hardware and software stacks. In that light, expressing specific number ranges as constraints would likely turn counterproductive.

Because our runtime system estimates the energy consumption and performance efficiency parameters at the software level model, the resulting estimations turn to be inaccurate. For example, the energy consumed by a method containing significant file I/O or using sensors may turn inaccurate because our energy model only takes CPU and network information into account. However, by adopting this energy measurement approach, we are trading potential inaccuracy for practicality. Because we aim at deploying our technology on standard consumer mobile devices, it would not be practical to use specialized hardware to measure the exact amount of consumed energy.

6 Related Work

Our approach is related to other complementary efforts that optimize mobile applications via cloud offloading. In addition, the server selection problem has been applied in other contexts to improve the QoS of distributed systems. Because these research areas are vast and extensive, we next compare and contrast our work only with the most closely related examples of prior work.

6.1 Optimizing Mobile Applications via Cloud Offloading

The cloud offloading optimization for mobile applications has been heavily covered in the research literature, with the following approaches sharing objectives or techniques with this work. CloneCloud [5] leverages hardware-based dynamic profiling to automatically partition a mobile application, enabling the server partition to migrate workloads at the thread level by means of a customized VM. ThinkAir [6] offloads energy intensive methods to the cloud, so that the resulting cloud-based execution can be scaled up by running the offloaded methods in parallel on dynamically allocated VMs.

Our prior contributions to cloud offloading [7, 9] also optimized energy consumption by reducing the amount of transferred program state via program analysis and driving the offloading via adaptive middleware. Here, we shift our focus on using cloud offloading to achieve flexible optimization objectives that consider multiple criteria configured by the end-user. To the best of our knowledge, this work is the first to leverage constraint solving to express the complex requirements of adaptive cloud offloading and evaluate them at runtime.

6.2 Server Selection

The problem addressed in this work is related to replica selection in distributed systems and service composition in service-oriented applications. Next, we briefly cover each research topic to explain how our approach differs from them.

Replica Selection In distributed systems, servers are replicated to improve robustness, scalability, and performance. Replica selection algorithms can be static or dynamic algorithms [13]. Static algorithms for load balancing assign replicas based on predefined rules (e.g., round robin, random access, proportional access, etc). Dynamic algorithms are used to improve the quality of service [4] as well as reduce overheads, increase accuracy, and support scalability.

Our approach differs in two ways. First, it executes the client code (at the method boundary) at a remote server rather than accessing the server's functionality. Second, it selects replicas adaptively, as based both on the end user's configuration and the runtime environment in place.

Web Service Composition Composing Web services is often driven by users, with complex service scenarios. Based on business processes, Web services are composed by using configuration files [2] or domain-specific languages [11] to express complex service requirements. The heterogeneous Web environment imposes the QoS challenges on publishing, locating, and invoking web services. Constraint solving has also been used to compose Web services efficiently. A constraint driven Web service composition framework METEOR-S [1] binds Web services and generates an executable process; it dynamically selects the best service candidate given the constraints of performance, cost, reliability, and availability. General constraint-based optimizations, especially CSP, can satisfy user preferences and QoS requirements in selecting services [3, 14].

Our approach applies constraint satisfaction to a new domain. Rather than composing software services, our approach selects a server to be used as a platform for executing portions of functionality of mobile applications. Another major difference is focusing our constraints-driven optimization on energy savings.

7 Conclusion

This paper introduced a novel dynamic adaptation approach, *constraints-driven adaptive offloading*, to optimizing mobile applications for energy-efficiency, performance, availability and privacy. The novelty of our approach lies in expressing this optimization problem in terms of constraint solving and providing an efficient runtime system that implements this adaptive offloading mechanism. As configurability has become an intrinsic requirement for modern software, our approach provides an expressive and efficient solution to the problem of adaptively leveraging cloud computing resources to optimize mobile applications.

References

1. R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven Web service composition in METEOR-S. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, 2004.

2. P. Albert, L. Henocque, and M. Kleiner. Configuration based workflow composition. In *Proceedings of the 2005 IEEE International Conference on Web Services*, 2005.
3. A. Ben Hassine, S. Matsubara, and T. Ishida. A constraint-based approach to horizontal Web service composition. In *Proceedings of the 5^{th} International Conference on The Semantic Web*, 2006.
4. R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of the IEEE 16^{th} Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1014–1021. IEEE, 1997.
5. B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the 6^{th} ACM European Conference on Computer Systems*, 2011.
6. 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 the IEEE Annual Joint Conference of the IEEE Computer and Communications Societies*, 2012.
7. Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32^{nd} International Conference on Distributed Computing Systems*, 2012.
8. Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Inf. and Software Technology*, 55(9):1602–1613, 2013.
9. Y.-W. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *Proceedings of the 29^{th} IEEE International Conference on Software Maintenance*, 2013.
10. Y. Li, H. Chen, and W. Shi. ACM HotMobile 2013 poster: Bugu: An application level power profiler and analyzer for mobile devices. *SIGMOBILE Mob. Comput. Commun. Rev.*, 17(3):27–28, Nov. 2013.
11. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, et al. OWL-S: Semantic markup for Web services. 2004.
12. N. Tamura, T. Tanjo, and M. Banbara. System description of a SAT-based CSP solver sugar. In *Proceedings of the 3^{rd} International CSP Solver Competition*, pages 71–75, 2009.
13. C. Tan and K. Mills. Performance characterization of decentralized algorithms for replica selection in distributed object systems. In *Proceedings of the 5^{th} International Workshop on Software and Performance*, 2005.
14. R. Thiagarajan and M. Stumptner. Service composition with consistency-based matchmaking: A CSP-based approach. In *Proceedings of the 5^{th} European Conference on Web Services*, 2007.
15. E. Tilevich and Y. Smaragdakis. NRMI: Natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):174–187, 2008.
16. A. Vahdat, A. Lebeck, and C. S. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 31–36. ACM, 2000.
17. L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8^{th} IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.