Optimal Predictive Code Offloading

Florian Berg
Institute of Parallel and Distributed Systems
University of Stuttgart
70569 Stuttgart, Germany
Florian.Berg@ipvs.uni-stuttgart.de

Frank Dürr
Institute of Parallel and Distributed Systems
University of Stuttgart
70569 Stuttgart, Germany
Frank.Duerr@ipvs.uni-stuttgart.de

Kurt Rothermel
Institute of Parallel and Distributed Systems
University of Stuttgart
70569 Stuttgart, Germany
Kurt.Rothermel@ipvs.uni-stuttgart.de

ABSTRACT

Modern mobile devices like smart phones and tablets are equipped with powerful processing and memory resources, enabling resource-intensive mobile applications such as high-end mobile games. The main limitation, however, remains the energy resource. To improve the energy efficiency, *code offloading* has been proposed, which offloads code to remote servers and transfers the results back to the mobile device. Although several approaches have shown that code offloading improves energy efficiency significantly in general, they largely neglect the adverse effects of network disconnections. Therefore, we have proposed the concept of *preemptive code offloading* to improve energy efficiency also under link failures. It transmits so-called safe-points between server and mobile device during remote execution, enabling the re-use of partial remote results after link failures. In this paper, we improve our basic preemptive code offloading approach by optimizing the time when to generate and transmit safe-points to minimize the communication overhead and maximize energy efficiency. To find the optimal safe-point schedule, we use a predictive approach that predicts the mobile link quality in order to send safe-points before network disconnections. Moreover, we consider additional deadline constraints for code execution to ensure a minimal responsiveness of offloaded applications despite link failures. Our evaluation results show that energy efficiency can be improved significantly using our predictive offloading approach.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/Server, Distributed Applications*

General Terms

Design, Experimentation, Measurement, Reliability

Keywords

Preemptable Code Offloading, Connectivity Prediction, User Movement, Mobile Cloud Computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MOBIQUITOUS 2014, December 02-05, London, Great Britain
Copyright © 2014 ICST 978-1-63190-039-6
DOI 10.4108/icst.mobiquitous.2014.258023

1. INTRODUCTION

Over the last decade, mobile devices such as smart phones and tablets have been equipped with increasingly powerful local resources like multi-core CPUs, GPUs, and several gigabyte of RAM. Similarly, wireless communication technologies got more and more powerful, enabling data rates of several megabit per second using LTE or WiFi at low monetary cost (“flat rates”). The availability of powerful mobile devices enables resource-intensive mobile applications such as mobile gaming or mobile office suites. Considering these improvements in processing power and memory, battery capacity is today the most limiting factor of mobile devices. Using resource-intensive applications, batteries are depleted in a few hours. Therefore, there is a strong incentive to improve the energy efficiency of mobile devices.

As a result, *code offloading* approaches have become popular. The basic idea of these approaches is to offload code dynamically at runtime from the mobile device to a remote server in the infrastructure and transfer the results of remote computations back to the mobile device. Utilizing the vast resources of modern cloud computing infrastructures, a large population of mobile devices can be served. Approaches like MAUI [4] or CloneCloud [3] have shown that energy efficiency can be improved by up to 45 %.

Although these approaches show the general potential of code offloading, they largely neglected the problems arising from unstable network connections. Intermittent failures of mobile network connections are still a significant problem in mobile communication networks as shown, for instance, in [2] and also our own experiments. For instance, while riding a local train in Stuttgart, Germany—a densely populated area with well-developed mobile communication infrastructure—, we could observer periods of several seconds without network connectivity. If not explicitly handled, these network failures have adverse effects on both the energy efficiency and user experience of code offloading. On the one hand, energy efficiency suffers since either offloaded code has to be re-executed locally after a timeout, or the mobile device has to wait for a re-connection wasting energy while waiting. On the other hand, the responsiveness of the often interactive mobile applications suffers since both re-execution or waiting for re-connection increase latency until the offloading result is available.

To alleviate these problems, we have presented the concept of *preemptable code offloading* in [1], explicitly tackling the

problem of communication failures. The basic idea of preemptable code offloading is to create so-called *safe-points* during remote execution, capturing the partial result of remote computation so far. These safe-points are sent to the mobile device. When a communication failure occurs, the mobile device can now re-use the partial results of remote execution, avoiding the complete re-execution of offloaded code. Although this concept introduces communication overhead for safe-point transmissions, we showed in [1] that in general it reduces energy consumption significantly and improves application latency under network failures.

Although [1] shows the benefits of preemptive code offloading in general, it uses a simple static strategy to determine the times for safe-point creation and transmission (safe-point schedule), which is not always optimal w.r.t. energy efficiency. Therefore, in this paper, we improve the basic approach by optimizing safe-point scheduling to minimize the communication overhead for transmitting safe-points. Obviously, for very stable communication networks, fewer safe-points are sufficient to reduce the communication overhead, whereas in unstable networks we should create more safe-points to be prepared for impending network failures. As an additional constraint, we want to guarantee a certain deadline for the overall code execution in order to guarantee a certain minimal application responsiveness. Thus, the problem is to design an *adaptive* safe-point scheduling algorithm, to adapt safe-pointing times to dynamic network conditions such that the given constrained optimization problem (minimum safe-pointing overhead under execution deadline constraints) is solved. To this end, we use a *predictive approach* to forecast the quality of mobile communication links.

In detail, we make the following contributions in this paper:

- A formulation of the problem of optimal safe-point scheduling under temporal disconnections and given execution time deadline constraints.
- An algorithm to solve this problem using Integer Linear Programming and a prediction model for mobile link quality.
- An implementation of the predictive code offloading approach and evaluation of its efficiency using simulations, calibrated through energy measurements on real mobile devices with different mobile applications and with real world connectivity traces.

Our measurements show that our predictive offloading approach reduces energy consumption significantly (up to 27%).

The rest of the paper is organized as follows. The next section describes the related work for our predictive code offloading approach. Afterwards, in Section 3, we describe the system model and problem formulation. Then, we describe in Section 4 our predictive code offloading approach in detail, and evaluate it in Section 5 on different mobile devices and applications. Finally, we conclude this paper.

2. RELATED WORK

Several code offloading approaches have been proposed recently. Cuervo et al. [4] propose an offloading approach called MAUI, which migrates code at the granularity of methods. To this end, it offloads annotated methods based on an optimization problem to minimize energy consumption of the mobile device. The CloneCloud system presented by Chun et al. [3] offloads threads from the mobile device to a remote server. One design goal of CloneCloud is to make offloading transparent to the application developer by automatically transforming and partitioning application code to enable a distributed execution. Code partitioning is optimized offline to minimize total execution time or energy consumption on the mobile device. The ThinkAir approach by Kosta et al. [5] additionally scales the computational power of the remote server dynamically to achieve optimal performance. All of these approaches show that code offloading can increase the energy efficiency significantly. However, they assume a permanent Internet connection and do not focus on the effects of link failures.

A code offloading approach explicitly handling link failures was proposed by Kwon et al. [6]. Whenever a link failure is detected, the remote code is re-executed locally. A second basic strategy to handle link failures is to wait for a reconnection to receive the results of offloading, as proposed, for instance, by Li et al. [7]. Both strategies might not be optimal w.r.t. energy efficiency and give no guarantees about execution deadlines.

Therefore, we proposed preemptive code offloading in [1], which enables the re-use of partial remote results in case of communication failures. Preemptive code offloading creates safe-points, capturing the temporal state of offloaded function on the remote server, and transmits these safe-points to the mobile device. After a link failure, the mobile device uses the last received safe-point to continue execution. However, so far, preemptive code offloading used a simple non-predictive safe-point scheduling strategy, which does not always perform optimally under dynamic network conditions.

3. SYSTEM MODEL AND PROBLEM STATEMENT

Next, we describe our system model and state the problem to be solved.

3.1 System Model

Our system consists of the following components, which are depicted in Figure 1: A battery-operated *mobile device*, an *offloading server*, and a *communication network*.

The battery-operated *mobile device* executes the *mobile application* in a *virtual machine* (VM) (a Java virtual machine (JVM) in our implementation). In order to support code offloading, we have extended a standard VM (Jikes RVM[1]) with our offloading framework to transfer application code and state between the mobile device and the offloading server during runtime as presented in the next section. We refer to the parts of the application state that are transfered between mobile device and server as *safe-points* (conceptually, a snapshot of the state of the offloaded code).

The *offloading server* executes the offloaded parts of the mobile application on behalf of the mobile device. The offload-

[1] `http://jikesrvm.org/`

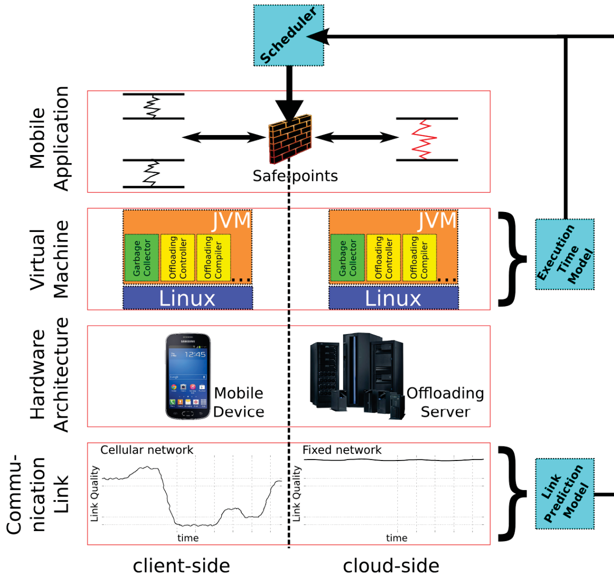

Figure 1: Overview of the system model.

ing server is a classic server machine, for instance, hosted in a cloud data center. It executes the same type of VM as the mobile device to abstract from the actual physical hardware. Also this VM executes our offloading framework to support code offloading. Moreover, the offloading server is executing the *safe-point scheduler*, which decides when to create and send safe-points to the mobile device. The scheduler uses a *link prediction model* to predict the availability of a mobile communication link between the mobile device and the Internet. Moreover, it uses an *execution time model* to estimate the execution time of code on the mobile device and server to meet execution time deadlines.

Mobile device and offloading server communicate via a wide-area *communication network*, consisting of a mobile communication network (e.g., cellular LTE network) and fixed network(s). We assume that wireless links of the mobile network might suffer transient failures, e.g., due to incomplete mobile network coverage, which might last for periods of several seconds or longer, depending, for instance, on the mobility of the mobile device. The mobile network is connected to the fixed network to provide Internet connectivity and connectivity to the offloading server. This fixed network is assumed to have negligible packet loss.

3.2 Problem Statement

The overall goal of our approach is to minimize energy consumption of the mobile device for code execution by offloading code to the offloading server and to guarantee a maximum time for code execution, both under network failures.

In more detail, we offload code at the granularity of functions (more precisely, Java methods). Thus, energy consumption is evaluated per method execution. The overall energy consumption E^M for executing method M is the sum of the following parts: (1) E^M_{local}: the energy for executing parts of M locally on the mobile device; (2) $E^M_{\text{offloaded}}$: the energy spent by the idle mobile device while M is executed on the server and the mobile device is waiting for the result (note that $E^M_{\text{offloaded}} > 0$ since also waiting consumes energy, e.g., for an active display or an idle communication interface); (3) $E^M_{\text{safepointing}}$: the energy for transferring safe-points between mobile device and server. Note that M can be offloaded to the server and loaded back onto the mobile device multiple times during the execution, thus $E^M_{\text{safepointing}}$ depends on the number of created safe-points. Thus, E^M is defined as:

$$\begin{aligned} E^{\text{M}} &= E^{\text{M}}_{local} &+ \\ &\quad E^{\text{M}}_{safepointing} &+ \\ &\quad E^{\text{M}}_{offloaded} \end{aligned} \quad (1)$$

Similarly, we calculate the execution time T^M of M as the sum of the local execution time T^M_{local}, remote execution time $T^M_{\text{offloaded}}$, and time for transferring safe-points $T^M_{\text{safepointing}}$:

$$\begin{aligned} T^{\text{M}} &= T^{\text{M}}_{local} &+ \\ &\quad T^{\text{M}}_{safepointing} &+ \\ &\quad T^{\text{M}}_{offloaded} \end{aligned} \quad (2)$$

Formally, the objective is to minimize E^M under a given maximum execution time constraint T^M_{max}:

$$\begin{aligned} \min \quad & E^{\text{M}} \\ s.t. \quad & T^{\text{M}} \leq T^{\text{M}}_{\text{max}} \end{aligned} \quad (3)$$

E^M and T^M are both influenced by the number of safe-points and time, when safe-points are created. Thus, the goal is to find an optimal *safe-point schedule* to solve the given constrained optimization problem under dynamic network conditions (link failures, varying network bandwidth).

4. PREDICTIVE CODE OFFLOADING

In the following section, we describe our predictive code offloading approach. We start with an overview of the offloading process, before we describe how to solve the problem of optimal safe-point scheduling. The solution is based on the prediction of different influencing factors such as connectivity or code execution time. At the end of this section, we present how we predict these values.

4.1 Overview

Before functions can be offloaded, the offloading framework has to identify feasible functions suitable for offloading. Typically, resource-intensive functions are good offloading candidates where server support reduces significantly execution time. To identify offloading candidates during runtime, we utilize a similar method as described in [4], supported by the application programmer, who annotates functions as `offloadable`. During runtime, our offloading framework automatically instruments the application's byte-code with breakpoints during just-in-time code compilation and monitors the resource usage of the instrumented code to define a profile of the function's code including execution times of code blocks (cf. Section 4.4). This profile together with monitored information about current network conditions serves as input to the safe-point scheduler, which decides whether a function is offloaded.

Whenever the safe-point scheduler decides to offload a candidate function to the offloading server, the offloading framework creates a first safe-point, containing the function parameters as well as all referenced data, and transmits an offloading request with the safe-point to the server. At the same time, a *failure detector* is started on the mobile device, which monitors incoming results (final or intermediate safe-points) from the server. Using the *execution time predictor* (cf. Section 4.4), the failure detector sets a deadline t_{maxwait} until which it must have received a safe-point from the server to meet maximum execution time constraint $T^M_{\max}$. t_{maxwait} is adaptively set such that a local execution starting at the last received safe-point will end before the maximum execution time. If no further safe-point is received from the server until t_{maxwait}, the mobile device starts a local execution based upon the state of the last received safe-point.

When the offloading server receives an offloading request with the first safe-point, it first instruments the byte-code of the offloaded function with breakpoints. These breakpoints mark places where safe-points can be created. Breakpoints are placed strategically at branching instructions like `if` or `loop` such that the memory footprint of safe-points is small. Branching instructions are particularly well-suited because, for instance, locally scoped variables of inner blocks do not need to be included into a safe-point.

Whenever the server hits a breakpoint during remote execution, the *safe-point scheduler* running on the server determines whether a safe-point should be created and sent to the mobile device. To this end, the scheduler solves the optimal safe-point scheduling problem under maximum execution time constraint (cf. Section 4.2) using Integer Linear Programming (ILP) as presented in detail in the next subsection. The solution of this problem is based on the *prediction of connectivity* as well as the *prediction of code execution time* as shown in Section 4.3 and Section 4.4, respectively. Intuitively, a safe-point should be generated and sent at the current breakpoint if the probability of connection loss until the next breakpoint is high. Otherwise, we can take the risk of waiting until the next breakpoint is reached, to again decide about safe-point creation. Thus, informally, the scheduler waits for the next breakpoint without sending a safe-point if either the probability of connection loss is low or the remaining duration of reaching the maximum execution time is high. Ideally, creating a safe-point at the server is equivalent to the decision to switch the execution side from server to mobile device, since the server only creates a safe-point if, according to the prediction models, the next breakpoint is not reachable due to the next network disconnection. So if the scheduler decides to send a safe-point to the mobile device, the mobile device continues the local execution after receiving the safe-point. Theoretically, the mobile device can decide to offload the function again to the server after the disconnection, although this is unlikely if network disconnections last longer than the remaining execution time of the function.

When the scheduler decides to create a safe-point, it generates a snapshot of the current function state by encoding only the delta to the last safe-point into the current safe-point to minimize safe-point size. Afterwards, it transmits the safe-point to the mobile device.

4.2 Optimal Safe-point Scheduling

The goal of the optimal scheduling is to find a safe-point schedule that minimizes the energy overhead under given maximum execution time constraints. In order to solve this scheduling problem, we model the execution of an offloaded function with the following *graph-based execution model.* The execution graph $G = (V, E, \varepsilon^V, \varepsilon^E, \tau^V, \tau^E)$ is a vertex-weighted and edge-weighted graph, consisting of a set of vertices V, where each $v \in V$ represents an *execution block.* An execution block is defined as the executed code between two breakpoints, which are inserted at branching instructions in the byte-code (cf. Section 4.1). An edge $e(u, v) \in E$ with $u, v \in V$ defines a transition between execution blocks according to the program structure. Figure 2 shows an exemplary graph with nine execution blocks:

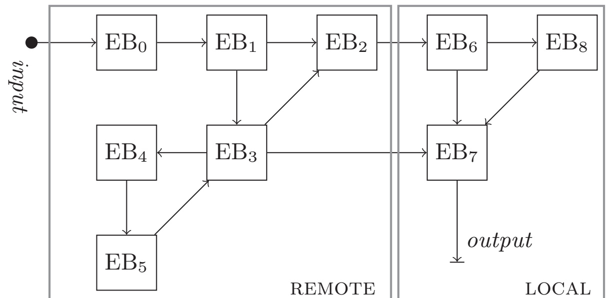

Figure 2: A sample execution graph for a method.

Function $\varepsilon^V : V \to \mathbb{R}$ defines a vertex weight that denotes the energy for executing the corresponding execution block. Function $\varepsilon^E : E \to \mathbb{R}$ defines an edge weight denoting the energy for (off-)loading the function from the mobile device to the server and vice versa between two blocks (i.e., switching the execution side).

Function $\tau^V : V \to \mathbb{R}$ defines the execution time of a block according to the execution time prediction model. Function $\tau^E : E \to \mathbb{R}$ defines the time for switching execution from the local to remote side and vice versa depending on the network bandwidth and latency.

In order to find the optimal safe-point schedule, we formulate the problem as an Integer Linear Program (ILP). Intuitively, we search for the *placement of each execution block* of the execution graph on the mobile device or the server, inducing the minimum energy cost and not violating the maximum execution time constraint. To define a placement of an execution block, we use a binary vector, $B \in \{0,1\}^{|V|}$, with $B_v = 0$ if block v is executed on the server and $B_v = 1$ if v is executed on the mobile device. Thus, the optimal solution of the ILP is the placement vector inducing the minimal energy cost without constraint violation.

Using this placement vector, we can express the induced energy cost by the following equation:

$$
\begin{aligned}
E^{\text{M}}(t) \quad &= \sum_{v \in V} \Big[B_v * \varepsilon_l^v + (1 - B_v) * (\tau_r^v * E^{\text{idle}}) \Big] + \\
&\sum_{e(u,v) \in E} |B_u - B_v| * \varepsilon^e_{B_u, B_v}
\end{aligned}
\tag{4}
$$

If an execution block is placed on the mobile device, the energy for local execution is added to the cost; if the block is executed remotely, the idle energy for waiting for the result is added. The second term defines the energy for switching execution sides, including safe-point creation and transmission. If the block placement is different for two linked blocks of the graph, we have to add the energy cost for switching.

Similarly, we can calculate the execution time T^{M} for a placement of execution blocks:

$$\begin{aligned} \sum_{v \in V} [B_v * \tau_l^v + (1 - B_v) * \tau_r^v] \quad + \\ \sum_{e(u,v) \in E} |B_u - B_v| * \tau^e_{B_u,B_v}(t) \quad \leq \quad T^{\mathrm{M}}_{\max} \end{aligned} \tag{5}$$

The first term defines the sum of local and remote execution times for all blocks. The second term calculates the time for switching sides of neighboring blocks.

Note that both energy cost and execution time depend on the individual execution block since different code blocks have different byte-code with different computational complexity. Moreover, execution time of the same block differs whether it is performed locally or remotely. Thus, we express energy cost and execution time as follows:

$$\begin{aligned} \tau^v_{l|r} &= J^v * f_T^{l|r} \\ \varepsilon^v_l &= J^v * f_E^l \end{aligned} \tag{6}$$

Here, J^v represents the number of byte-code instructions for block v. f_E and f_T are device-specific factors, determining the energy and time for byte-code execution, where f_E is calibrated through offline benchmarking and f_T through online measurements.

The cost for switching the execution side is determined based on the data size of a safe-point, which depends on the modified variables since the last transmitted safe-point (static, class, and stack variables). For instance, a remote-to-local switch between blocks EB_2 and EB_6 in Figure 2 requires all modified values since the remote method start at node *input* and after node EB_2. Figure 3 shows an example of modifications at each execution block, resulting in an execution state size of ten bytes for the `int`, `float`, and `short` value.

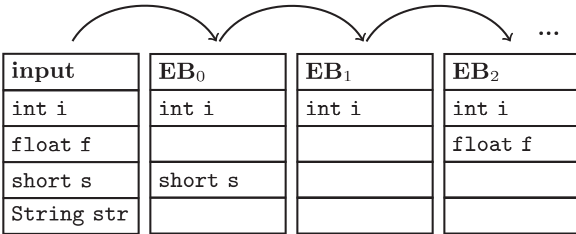

Figure 3: An exemplary modification of values within each execution block.

The energy cost and execution time for each $e = (u, v) \in E$ are determined based on the last synchronized execution state S_{i-1} with $i \geq 1$, where S_0 equals the execution state at the remote start of the offloaded function:

$$\begin{aligned} \tau^e_{B_u,B_v}(t) &= \delta(S_i, S_{i-1}) * f_{\mathrm{BW}}^{l\to r|r\to l}(t) + f_{\mathrm{LAT}}^{l\to r|r\to l}(t) \\ \varepsilon^e_{B_u,B_v} &= \delta(S_i, S_{i-1}) * f_{\mathrm{E}}^{l\to r|r\to l} \end{aligned} \tag{7}$$

Here, f_{BW} represents the current bandwidth at time t, f_{LAT} the current latency at time t, and f_{E} the energy cost for sending and receiving data. The resulting Integer Linear Program is defined as follows:

$$\begin{aligned} \min \quad & \sum_{v \in V} B_v * J^v * f_E^l + (1 - B_v) * J^v * f_T^r * E^{\mathrm{idle}} \\ & + \sum_{e(u,v) \in E} |B_u - B_v| * \delta(S_u, S_v) * f_{\mathrm{E}}^{B_u B_v} \\ s.t. \quad & \sum_{v \in V} B_v * J^v * f_T^l + (1 - B_v) * J^v * f_T^r \\ & + \sum_{e(u,v) \in E} |B_u - B_v| * \Big(\delta(S_u, S_v) * f_{\mathrm{BW}}^{B_u B_v}(t) \\ & \qquad + f_{\mathrm{LAT}}^{B_u B_v}(t)\Big) \quad \leq \quad T^{\mathrm{M}}_{\max} \end{aligned} \tag{8}$$

In order to solve the given ILP, standard solvers such as IBM CPLEX or the Matlab solver can be used on the server. These solvers are highly optimized and as our evaluations show can solve the given problem in a few milliseconds.

4.3 Connectivity Prediction

The optimal solution of the ILP depends on the knowledge of the future link bandwidth and latency denoted by variables f_{BW} and f_{LAT} in the above ILP. To define these variables, we use a connectivity prediction model based on Markov chains, characterized by low space and computational complexity. Markov chains are widely used in reserach on wireless link prediction, e.g., [10]. To keep the required (energy) overhead minimal for the connectivity model calibration, the approach utilizes bandwidth and latency values of any wireless connection from the mobile device to the Internet. No additional measurements have to be started in parallel. The continuous values for the latency and bandwidth are mapped onto n discrete *quality classes*. For instance, the five latency classes $[0\ ms, 50\ ms]$, $(50\ ms, 100\ ms]$, $(100\ ms, 200\ ms]$, $(200\ ms, 300\ ms]$, and $(300\ ms, \infty]$ could be interpreted as "very low latency", "low latency", "medium latency", "high latency", and "disconnected". Similarly, we define different classes for link bandwidth, e.g., $[0\ kbps, 50\ kbps]$, $(50\ kbps, 350\ kbps]$, $(350\ kbps, 700\ kbps]$, $(700\ kbps, 1050\ kbps]$, and $(1050\ kbps, \infty]$.

Each link direction (downstream and upstream) of the wireless channel is modeled as a time-continuous (time-homogeneous) Markov chain, where the finite state space $Q^{\mathrm{LAT}}_{\mathrm{BW}}$ consists of n^2 states. For instance, for the above link latency and bandwidth classes, $Q^{\mathrm{LAT}}_{\mathrm{BW}}$ consists of 25 states.

Variable p_{ij} with $i, j = [0, 1, \ldots, s]$ and $s = n - 1$ represents the transition probabilities between a state $Q^{\mathrm{LAT}_f}_{\mathrm{BW}_e}$ and a state $Q^{\mathrm{LAT}_h}_{\mathrm{BW}_g}$ with $e, f, g, h = [0, 1, \ldots, s]$, resulting in the

following transition rate matrix P:

$$P = \begin{pmatrix} p_{00} & p_{01} & \dots & p_{0s} \\ p_{10} & p_{11} & \dots & p_{1s} \\ \vdots & \vdots & \ddots & \vdots \\ p_{s0} & p_{s1} & \dots & p_{ss} \end{pmatrix} \quad (9)$$

The time duration spent in each state is a non-negative value partitioned into parts $t \in \{1, 2, \dots\}$ (e.g., with a time step size of 500 ms). The probability of a link failure and a corresponding link recovery are considered to be exponentially distributed. Furthermore, we assume that the Markov property is fulfilled (cf. Nicholson et al. [8]), i.e., the conditional probability distribution of a future state only depends on the current state x_t:

$$\begin{aligned} \mathrm{P}(X_{t+1} = x_{t+1} \mid\ & X_t = x_t, \\ & X_{t-1} = x_{t-1}, \\ & \dots, \\ & X_0 = x_0) \\ = \mathrm{P}(X_{t+1} = x_{t+1} \mid\ & X_t = x_t) \end{aligned} \quad (10)$$

Based on the transition rates, we calculate iteratively the mean recurrence time M_{ii} as well as the mean first passage time M_{ij}. M_{ij} represents the expected number of steps the stochastic process takes to reach node j from node i for the first time:

$$M^{(k+1)} = E + P * (M^{(k)} - diag\{M^{(k)}\}) \quad (11)$$

with

$$M^{(k)} = \begin{pmatrix} m_{00}^{(k)} & m_{01}^{(k)} & \dots & m_{0s}^{(k)} \\ m_{10}^{(k)} & m_{11}^{(k)} & \dots & m_{1s}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ m_{s0}^{(k)} & m_{s1}^{(k)} & \dots & m_{ss}^{(k)} \end{pmatrix} \text{ and}$$

$$diag\{M^{(k)}\} = \begin{pmatrix} m_{00}^{(k)} & 0 & \dots & 0 \\ 0 & m_{11}^{(k)} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & m_{ss}^{(k)} \end{pmatrix}$$

where $M^{(0)} = E$ and E denote the identity matrix [9].

Based on this model, the offloading server can forecast the two time-dependent connectivity factors f_{BW} and f_{LAT}, resulting in time-invariant factors by summing up the connected values for latency and bandwidth:

$$\begin{aligned} f_{\mathrm{BW}}^{l\to r|r\to l} &= \sum_{c,l} m_{cl}^{\infty} \\ f_{\mathrm{LAT}}^{l\to r|r\to l} &= \sum_{c,b} m_{bc}^{\infty} \end{aligned} \quad (12)$$

4.4 Execution Time Prediction

Finally, to satisfy the maximum execution time constraint, we need to predict the remaining local and remote execution time of offloaded functions, which is the objective of the execution time prediction model. The remaining local or remote execution time of an already partially executed function is calculated based on the individual execution times for remaining code blocks (cf. Equation 6).

To predict the execution path of a function (remaining code blocks), we use a history-based approach like the one used by MAUI [4]. This approach is based on a database of past function invocations, mapping input parameters to the resulting execution path. The remaining execution time can then be determined based on the execution times of the resulting code blocks.

5. EVALUATION

In this section, we evaluate the energy-efficiency of our predictive code offloading approach. Since the performance of our approach depends on various parameters such as network connectivity, energy characteristics of device, or application code, we use simulations to be able to evaluate the performance under various parameter combinations. To achieve realistic results, we calibrate these simulations using energy and connectivity measurements with different types of real devices in real environments. Next, we describe our simulation setup, before the evaluation results are presented.

5.1 Setup of Experiments

We consider two heterogeneous classes of mobile devices: a resource-constrained netbook and a more powerful laptop. The netbook is a Dell Inspiron Mini 10v with an Intel Atom N270 processor (1.6 GHz) and one GByte of RAM. The laptop is a Lenovo ThinkPad T61 with an Intel Core 2 Duo T7300 processor (2.0 GHz) and three GByte of RAM. Both mobile devices are equipped with a 3G communication interface (HUAWEI HSPA USB Stick; Model: E1750) for mobile Internet connectivity. As offloading server, we use a PC server with an Intel Core i7-2600 Quad-Core processor (3.4 GHz) and eight GByte of RAM.

We consider two Java-based applications that differ significantly w.r.t. computational complexity and communication overhead for offloading. The first application is a *chess game*, where the function for calculating the next move is an ideal offloading candidate with high computational complexity but small size of the function parameters, small safe-points, and a small result size. The second application is a *face-recognition* application. Here, the parameters of offloaded functions are large (e.g., image), however, safe-points and results are small.

As simulation tool, we use MATLAB. Simulations take into account various timing parameters as shown in Table 1, which basically depend on the processor speed of the device, complexity of the application code, and data rate of the communication link. T_{lsc} is the time until reaching the function to be offloaded. T_{rmc} and T_{lec} are the durations to execute the function remotely on the offloading server and locally on the mobile device, respectively. T_{lsg} and T_{lst} are the durations to generate and transfer the local state to the offloading server; T_{rsg} and T_{rst} are the similar durations for generating safe-points at the remote server and sending it to the mobile device. T_{lsi} and T_{rsi} are the required durations to install safe-points on the mobile device or remote server. Finally, T_h defines the maximum acceptable execution time for the applications, which we set lower for the interactive chess game than for the face recognition. We chose T_h based on the local application execution times on the corresponding mobile device.

Table 1: Simulation parameters

Name	Description
T_h	Application-specific time threshold
t_s	Simulation time step granularity
T_{lsc}	Time for reaching method to offload
T_{lmc}	Time for local method execution
T_{rmc}	Time for remote method execution
T_{lec}	Time for reaching application end
T_{lsg}	Generation time for a local state
T_{lst}	Transmission time of local state to the cloud
T_{lsi}	Local installation time for remote state
T_{rsg}	Generation time for a remote state
T_{rst}	Transmission time of remote state to device
T_{rsi}	Remote installation time for local state

The values of these (resource-dependent) simulation parameters, as shown in Table 2, were calibrated by executing both applications on a jRVM with a full implementation of our offloading framework on both of the physical devices. Moreover, during the same execution we measured the energy required for executing the code locally and remotely and for communicating safe-points. These measurements of timing and energy parameters on real physical devices provide a realistic basis for our simulation. Furthermore, we measured the overhead of the server-side optimizer by instrumenting it to record the time it takes to solve the ILP problem. The solver takes less than 50 ms for the chess game as well as for the face recognition application. Thus, the ILP can be solved quickly and keeps the introduced overhead small at the server.

In order to also base our simulation on realistic mobile network connectivity settings, we measured network connectivity in real cellular networks while moving on different days in Stuttgart, Germany. The connectivity traces were gathered during a train ride (where a user might play the chess game) using periodic ping messages over two cellular networks of the providers O_2 and T-Mobile to a server connected to the Internet through the fixed university network. For these traces, we generated the Markov models as described in Section 4.3 (T-Mobile and O_2 connectivity model), to model network connectivity. For instance, we get the following exemplary transition rates and mean recurrence/first passage times for the T-Mobile (Equation 13) and O_2 (Equation 14) cellular network, differing significantly in the number and duration of disconnections:

$$P = \begin{pmatrix} 0.8861 & 0.1139 \\ 0.1950 & 0.8051 \end{pmatrix} \; M_{ij}^{(\infty)} = \begin{pmatrix} 1.5842 & 8.7826 \\ 5.1304 & 2.7119 \end{pmatrix} \quad (13)$$

$$P = \begin{pmatrix} 0.9859 & 0.0141 \\ 0.1081 & 0.8919 \end{pmatrix} \; M_{ij}^{(\infty)} = \begin{pmatrix} 1.1307 & 70.7500 \\ 9.2500 & 8.6486 \end{pmatrix} \quad (14)$$

Informally, cellular link quality is higher for T-Mobile in comparison to O_2 (cf. $M_{01}^{(\infty)}$ of Equation 13 and 14). Due to the significance of the connectivity prediction, we evaluated the accuracy of the applied Markovian connectivity model. To this end, we divided the real-world connectivity traces into a training set and a test set. Afterwards, the Markov model is created based on the training set and evaluated against the test set. In average, the link quality is predicted properly in 65 % by the simple stochastic model.

Table 2: Calibrated timing parameters

Name	Chess		FaceRec	
	Netbook	Laptop	Netbook	Laptop
T_h	40.000 s	10.000 s	50.000 s	30.000 s
t_s	0.001 s			
T_{lsc}	5.797 s	1.421 s	2.903 s	1.684 s
T_{lmc}	27.705 s	7.307 s	39.158 s	21.728 s
T_{rmc}	4.249 s		6.594 s	
T_{lec}	0.778 s	0.110 s	5.806 s	3.261 s
T_{lsg}	1.146 s	0.655 s	3.182 s	1.746 s
T_{lst}	0.100 s		3.385 s	
T_{lsi}	0.157 s	0.014 s	0.128 s	0.015 s
T_{rsg}	0.029 s		0.024 s	
T_{rst}	0.010 s		0.009 s	
T_{rsi}	0.327 s		0.083 s	

We compare our optimized offloading approach denoted as *Predictive* in the following with four other approaches, denoted as *Local*, *Baseline*, *Periodic*, and *Optimal*. *Local* executes the complete application on the mobile device without offloading. In order to compare to related offloading approaches using only basic communication error handling without safe-pointing, we compare to the *Baseline* approach. This approach re-executes the offloaded function locally as soon as a disconnection is detected. This strategy is used, for instance, in [6]. *Periodic* uses a static (i.e., non-predictive and non-optimized) safe-pointing strategy, where the mobile device receives safe-points after an application-specific predefined time interval (1 s in our evaluation). *Optimal* serves as a reference with the perfect safe-pointing schedule, which can only be achieved theoretically, knowing in advance when disconnections happen and how long they will last.

5.2 Results: Mobile Chess Game

First, we present the results for the energy consumption and execution times of the mobile chess game. In these simulations, the maximum execution time constraint for the netbook is set to 40 s and for the laptop to 10 s due to the hardware heterogeneity of the mobile devices.

Figure 4 shows a histogram of the execution times for the netbook, wherefore 1278 T-Mobile connectivity traces are evaluated. As can be seen from the histogram, the execution times of all approaches are centered around two peeks at 12.60 s and 34.28 s. The first peek corresponds to runs where no disconnections occur and the function is executed completely remotely. 34.28 s corresponds to a fully local execution on the mobile device. The difference between these two values shows the general speed up of code offloading, utilizing the faster server hardware instead of the slower local processing resources (*Local* approach).

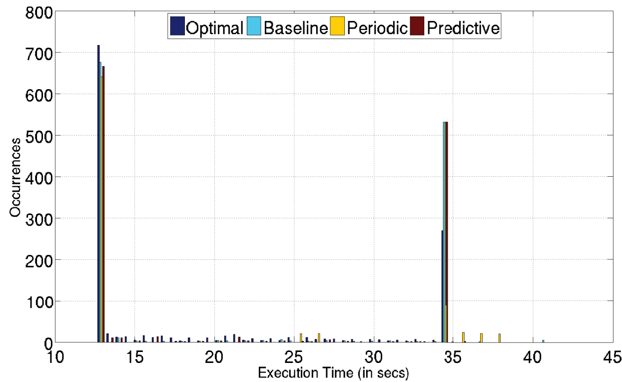

Figure 4: Netbook: Execution times for mobile chess game based on T-Mobile connectivity traces.

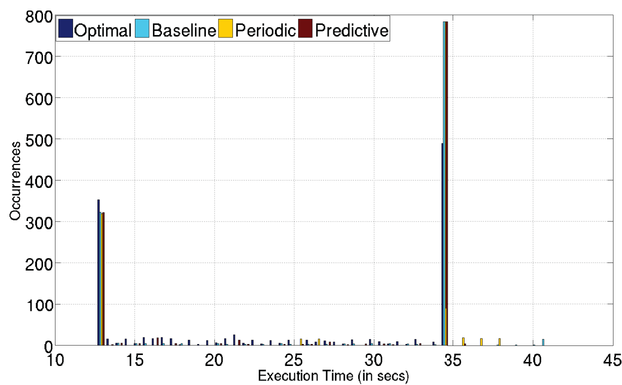

Figure 5: Netbook: Execution times for mobile chess game based on O_2 connectivity traces.

In comparison to the *Baseline* offloading approach, *Predictive* leads to slightly smaller execution times (22.08 s on average for *Predictive* vs. 22.25 s for *Baseline*). Moreover, *Baseline* violates the execution deadline five times when disconnections happen, whereas *Predictive* never violates the execution deadline. Like *Predictive*, *Periodic* never violates the deadline, showing the positive effect of safe-pointing in general. On average, *Periodic* has an execution time of 23.27 s, i.e., 5.4% higher than *Predictive*. Compared to *Optimal*, *Predictive* leads to 13% higher execution times on average.

For the O_2 cellular network, Figure 5 shows the corresponding histogram of the execution times on the netbook, where 1185 connectivity traces are evaluated. Here, the execution times of all approaches are also centered around the two peeks, however, 34.28 s dominates over 12.60 s. The main reason for this is that disconnections are more likely for the O_2 cellular network, wherefore the error handlers of *Baseline*, *Periodic*, and *Predictive* start local execution in case of an error. As a result, the execution times of all three approaches are increased heavily, e.g., *Baseline* by 26%, *Periodic* by 23%, and *Predictive* by 25%. This general increase of the execution times for the O_2 connectivity traces, however, results in the same characteristics as for the T-Mobile connectivity traces.

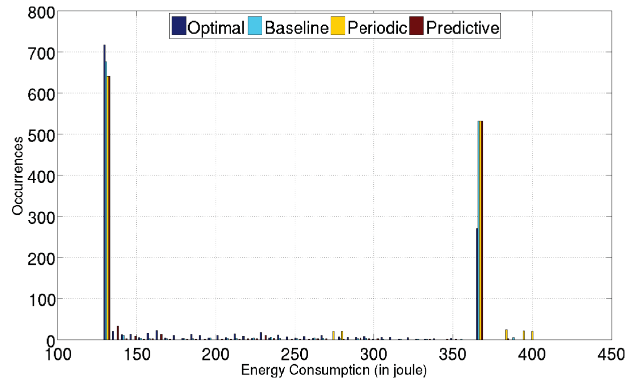

Figure 6: Netbook: Energy consumptions for mobile chess game based on T-Mobile connectivity traces.

On average, *Predictive* leads to smaller execution times than *Baseline*, which also violates the execution deadline, now, for 16 traces. *Predictive* stays for all traces within the execution deadline, just as *Periodic*, emphasizing the benefits of safe-pointing. In comparison to *Optimal*, the execution time of *Predictive* is 12% higher on average.

For the energy efficiency, Figure 6 shows a histogram of the energy consumption on the netbook for the T-Mobile connectivity traces, where also all approaches are mainly settled around two peeks, 128.53 J and 369.12 J. Here, 128.53 J corresponds to runs where no disconnections occur and 369.12 J to a fully local execution on the mobile device, resulting in energy savings of 240.59 J due to code offloading.

In comparison with *Baseline*, *Predictive* leads to slightly lower energy consumptions (233.95 J on average for *Predictive* vs. 234.43 J for *Baseline*). For *Periodic*, however, the energy savings of *Predictive* are higher, saving up to 13 J on average, due to fewer safe-point transmissions. Compared to *Optimal*, *Predictive* consumes 15% more energy on average.

The energy results for the O_2 cellular network are proportionally similar—only increased in general by the higher execution times—and are omitted due to the lack of space.

Now, we regard the execution times (cf. Figure 7 and 8) as well as energy consumptions (cf. Figure 9) for the T-Mobile and O_2 cellular network on the laptop. Due to relaxing the resource limitation, the time and energy results for both cellular networks are more closer to each other.

Figure 7 depicts the execution times for 1428 T-Mobile connectivity traces. Here, the two peeks are situated at 6.92 s and 8.84 s, meaning an offloading-related time decrease of 1.92 s. As a result, code offloading is harder for the laptop, since the offloading-related benefits are in total small.

For the laptop, *Predictive* leads to smaller execution times in comparison to *Baseline* as well as *Periodic* (on average 7.78 s for *Predictive* vs. 7.85 s for *Baseline* vs. 7.93 s for *Periodic*). Moreover, *Baseline* and also *Periodic* violate the execution deadline 56 and 74 times, whereas *Predictive* never violates it, showing the positive effect of utilizing prediction models.

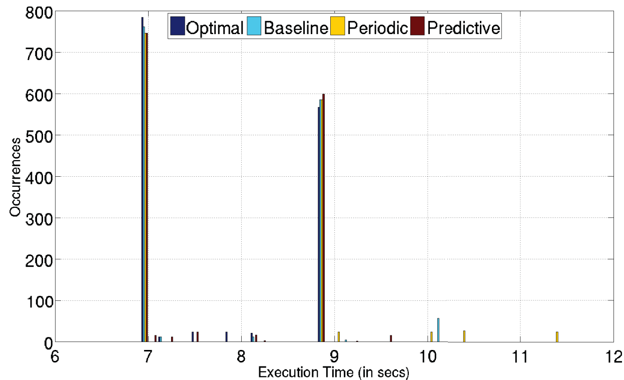

Figure 7: Laptop: Execution times for mobile chess game based on T-Mobile connectivity traces.

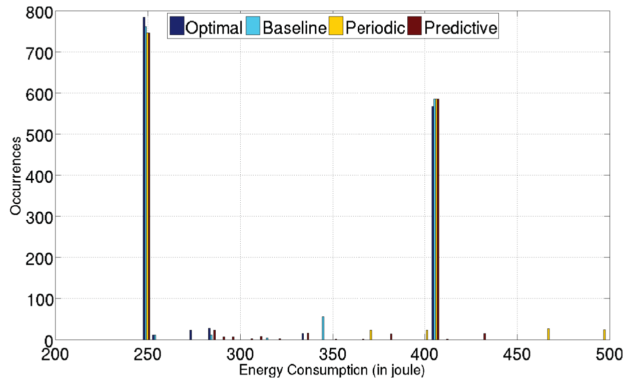

Figure 9: Laptop: Energy consumptions for mobile chess game based on T-Mobile connectivity traces.

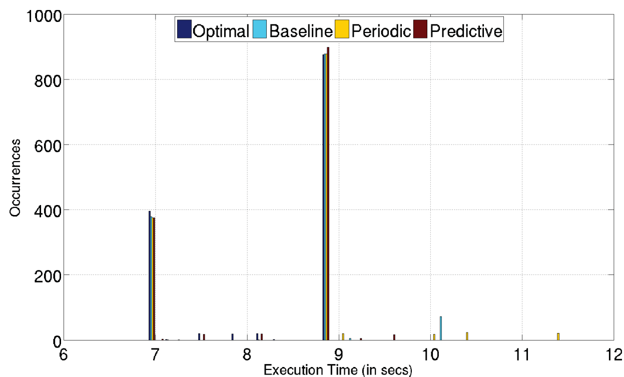

Figure 8: Laptop: Execution times for mobile chess game based on O_2 connectivity traces.

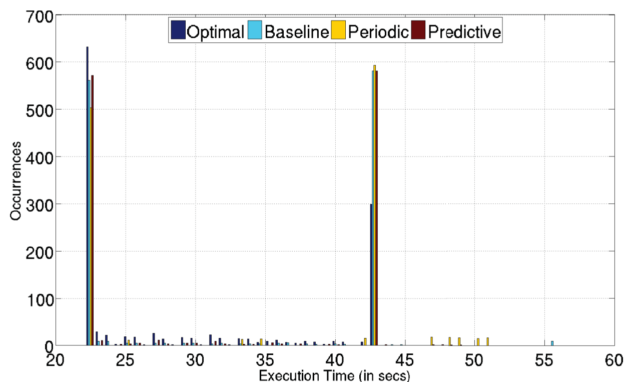

Figure 10: Netbook: Execution times for face recognition based on T-Mobile connectivity traces.

Compared to *Optimal*, *Predictive* leads only to 0.76% higher execution times on average, i.e., being very close to optimal.

Just like the netbook results, the execution times for the O_2 connectivity traces are also increased on the laptop in general (cf. Figure 8), mainly caused through the higher disconnection likelihood. As a consequence, the results are shifted towards the *Local* execution time peek, since error handling starts for all three approaches local execution in case of an error. On average, the execution times of *Baseline*, *Periodic*, and *Predictive* are all increased by roughly 2%. As a result, the above-described relation between the three approaches stays the same. Thus, *Baseline* as well as *Periodic* violates the execution deadline 72 and 62 times, whereas *Predictive* never violates it. Compared to *Optimal*, the execution time of *Predictive* is also only slightly higher on average (0.69%).

The energy consumptions for the T-Mobile and O_2 cellular networks can be summarized by the execution time characteristics. Figure 9 shows exemplarily the histogram of the energy consumptions for the T-Mobile connectivity traces. The energy results for the O_2 connectivity traces look similar, shifted towards the energy consumption of *Local* due to the higher likelihood of disconnections.

Compared to *Baseline*, *Predictive* leads to only somewhat lower energy consumptions of 0.1% for T-Mobile and 0.5% for O_2 on average. For *Periodic*, the energy savings are higher, resulting in 2.2% for T-Mobile and 1.4% for O_2 on average. However, in comparison with *Optimal*, the energy consumption of *Predictive* is only slightly higher for both networks (1.9%; 1.3%).

In summary, for the chess application, the *Predictive* approach shows better performances than the basic offloading approaches, without violating an execution time threshold despite temporary disconnections.

5.3 Results: Mobile Face Recognition

For the mobile face recognition application, Figure 10 shows a histogram of the netbook's execution times for the T-Mobile connectivity traces, whereas Figure 11 shows the laptop's execution times for the O_2 traces. The maximum execution time threshold is now set to 50 s for the netbook and 30 s for the laptop. In the following, we have summed up the results related to the T-Mobile and O_2 connectivity traces, since they are comparatively similar.

The above-described basic trend is similar for the mobile face recognition application. Our *Predictive* approach always meets the deadline, in contrast to the basic offloading approaches *Baseline* and *Periodic*. In detail, *Baseline* and

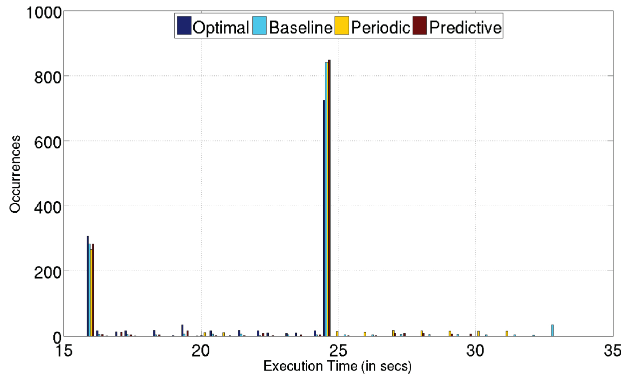

Figure 11: Laptop: Execution times for face recognition based on O_2 connectivity traces.

Periodic violate the given deadline in 1% and 2% of the cases when using the netbook as well as 3% and 3% of the cases when using the laptop. Moreover, *Predictive* always saves energy compared to a local execution or the basic offloading approaches. For the netbook, we save on average 18.17% compared to *Local*, 0.77% compared to *Baseline*, and 4.35% compared to *Periodic*; for the laptop, we save on average 13.55% compared to *Local*, 1.33% compared to *Baseline*, and 3.16% compared to *Periodic*.

In comparison with *Optimal*, *Predictive* consumes 10.6% more energy and takes 9.9% longer on average for the netbook, and consumes 6.6% more energy and takes 3.9% longer on average for the laptop.

So, in conclusion, our *Predictive* approach performs significantly better than the basic offloading approaches, often tending the optimal performance. The benefits are more pronounced for computational intensive applications with smaller state (safe-points) such as the considered chess game, which naturally lend themselves better to offloading than applications with larger state to be transferred between a mobile device and a server.

6. SUMMARY

Code offloading promises to increase the energy-efficiency of resource-intensive mobile applications by offloading code at runtime to a remote server. Although existing approaches have shown the general effectiveness of code offloading, they largely neglected the practical problem of network disconnections between mobile device and server, which have adverse effects on the energy-efficiency of offloading. To solve this problem, we have presented a novel code offloading approach in this paper based on two concepts: Firstly, we use the concept of *safe-points*, which capture a partial result of offloaded code. By transferring safe-points between offloading server and mobile device, the mobile device can benefit from partial results calculated by the server until the last safe-point before a disconnection. Secondly, we presented an *optimal safe-point scheduling algorithm* that optimizes energy efficiency under maximum execution time constraints by selecting the optimal time to send safe-points. To this end, we used a predictive approach taking into account future network connectivity. Our evaluation results show that this approach increases energy efficiency significantly compared to basic offloading approaches, and additionally guarantees maximum execution times under network failures.

7. ACKNOWLEDGMENTS

This work was partially funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

8. REFERENCES

[1] F. Berg, F. Dürr, and K. Rothermel. Increasing the Efficiency and Responsiveness of Mobile Applications with Preemptable Code Offloading. In *Proc. 3rd Intl. Conf. Mobile Services*, pages 76–83. MS'14, June 2014.

[2] Y.-C. Chen, E. M. Nahum, R. J. Gibbens, and D. Towsley. Measuring Cellular Networks: Characterizing 3G, 4G, and Path Diversity. In *Annual Conference of International Technology Alliance*, ACITA'12, 2012.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. 6th Conf. Computer Systems*, EuroSys'11, pages 301–314, 2011.

[4] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. 8th Intl. Conf. Mobile Systems, Applications, and Services*, MobiSys'10, pages 49–62, March 2010.

[5] 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 *Proc. IEEE INFOCOM 2012*, pages 945–953, March 2012.

[6] Y.-W. Kwon and E. Tilevich. Energy-Efficient and Fault-Tolerant Distributed Mobile Execution. In *IEEE 32nd Intl. Conf. Distributed Computing Systems*, pages 586–595, June 2012.

[7] J. Li, K. Bu, X. Liu, and B. Xiao. ENDA: Embracing Network Inconsistency for Dynamic Application Offloading in Mobile Cloud Computing. In *Proc. 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC'13, pages 39–44, August 2013.

[8] A. J. Nicholson and B. D. Noble. BreadCrumbs: Forecasting Mobile Connectivity. In *Proc. 14th ACM Intl. Conf. on Mobile Computing and Networking*, MobiCom'08, pages 46–57, September 2008.

[9] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, New Jersey, USA, 1994.

[10] W. Zhang, Y. Wen, K. Guan, D. Kilper, H. Luo, and D. Wu. Energy-Optimal Mobile Cloud Computing under Stochastic Wireless Channel. *Wireless Communications, IEEE Transactions on*, 12(9):4569–4581, September 2013.