Increasing the Efficiency of Code Offloading through Remote-side Caching

Florian Berg, Frank Dürr, Kurt Rothermel
Institute of Parallel and Distributed Systems
University of Stuttgart
70569 Stuttgart, Germany
Email: {Berg, Duerr, Rothermel}@ipvs.uni-stuttgart.de

***Abstract*—End users execute today on their smart phones different kinds of mobile applications like calendar apps or high-end mobile games, differing in local resource usage. Utilizing local resources of a smart phone heavily, like playing high-end mobile games, drains its limited energy resource in few hours. To prevent the limited energy resource from a quick exhaustion, smart phones benefit from executing resource-intensive application parts on a remote server in the cloud (*code offloading*). During the remote execution on the remote server, a smart phone waits in idle mode until it receives a result.**

However, code offloading introduces computation and communication overhead, which decreases the energy efficiency and induces monetary cost. For instance, sending or receiving execution state information to or from a remote server consumes energy. Moreover, executing code on a remote server instance in a commercial cloud causes monetary cost. To keep consumed energy and monetary cost low, we present in this paper the concept of remote-side caching for code offloading, which increases the efficiency of code offloading. The remote-side cache serves as a collective storage of results for already executed application parts on remote servers, avoiding the repeated execution of previously run application parts. The smart phone queries the remote-side cache for corresponding results of resource-intensive application parts. In case of a cache hit, the smart phone gets immediately a result and continues the application execution. Otherwise, it migrates the application part and waits for a result of the remote execution. We show in our evaluation that the use of a remote-side cache decreases energy consumption and monetary cost for mobile applications by up to 97% and 99%, respectively.

***Keywords**:* Mobile Cloud Computing, Code Offloading, Distributed Execution, Data Replication, Function Caching

I. Introduction

Nowadays, smart phone manufacturers equip their products with high-end system-on-a-chip (SoC) hardware like the octa-core CPU Snapdragon 810 from Qualcomm (up to 2 GHz). Such high-end SoCs support also high-bandwidth wireless communication like 4G LTE or WiFi, to provide fast mobile access to the Internet for end users. End users execute on the local hardware different kind of mobile applications like playing high-end mobile games or obtaining up-to-date information from Facebook or Twitter. Despite the powerful SoCs of modern smart phones, the battery capacity is still the most limiting resource, in particular, for the execution of resource-intensive applications. Hence, the main limitation of today's smart phones remains the constrained battery capacity.

One approach to handle the constrained battery capacity is code offloading. Code offloading *identifies* resource-intensive application parts of applications and *migrates* them to a remote server in the cloud. The remote server then executes on its virtually unlimited resources the migrated application parts, while the smart phone *waits* in idle mode for a corresponding result. After the smart phone *receives* the result of a migrated application part from the remote server, it continues the application execution on its local hardware. As a result, a smart phone saves energy with code offloading, if the offloading process for an application part – identify, migrate, wait, and receive – consumes less energy than a corresponding local application part execution (up to 45%; cf. [1] or [2]).

To identify application parts for code offloading, an offloading algorithm considers the tradeoff between local execution cost and remote execution cost. Local execution cost for an application part includes energy for executing it on the local hardware. Remote execution cost for an application part includes energy and additionally money. Migrating, waiting, and receiving consume energy. Executing the migrated application part on a server instance in a commercial cloud causes a monetary cost, typically in a pay-as-you-go manner. As a result, two properties define an ideal application part for code offloading: First, a long running computation, where the speed-up of the cloud reduces execution time; second, a small input and output size of the application part, why transmission overhead is low. Because a long running computation on a smart phone also takes some time on a remote server, the energy consumption on the mobile device as well as the monetary cost for the cloud instance are not negligible. Both reduce the overall acceptance of code offloading approaches.

To handle these problems, we investigate the collective sharing of results from already migrated application parts in a remote-side cache. After a smart phone has identified a suitable application part for a remote execution, it first queries the remote-side cache for a corresponding result. In case of a cache hit, the smart phone receives immediately the result and continues the application execution. In case of a cache miss, it migrates the resource-intensive application part to a server in the cloud and waits until it receives a result. Thus, our approach benefits from application executions on different smart phones, where remote servers store the results of migrated application parts in the same remote cache. As a result, the integration of a remote-side cache to code offloading replaces the offloading time cost with the caching space cost.

978-1-4673-7701-0/15/$31.00 ©2015 IEEE

In detail, we make the following contributions:

- A problem formulation for the integration of a remote-side cache to code offloading
- An algorithm to solve the caching-enabled offloading problem on a mobile device
- The implementation of the caching-enabled code offloading algorithm
- An evaluation of its efficiency based on real mobile devices with different mobile applications

Our measurements show that our caching-enabled code offloading approach reduces energy consumption by up to 97% on the mobile device and monetary cost by up to 99%.

The rest of the paper is organized as follows. The next section describes the related work for our caching-enabled code offloading approach. Afterwards, in Section III, we describe the system model and problem formulation. Then, we describe in Section IV our caching-enabled code offloading approach in detail, and evaluate it in Section V on different mobile devices and applications. Finally, we conclude our paper and give a short outlook on future work.

II. Related Work

In this section, we describe related work, categorized into two groups: data/function caching and code offloading.

Today, many computing systems utilize different types of data/function caches. For instance, nodes in a distributed system utilize a DNS cache for speeding up name resolution. The basic idea behind the caching concept is that a cache stores information of current requests, speeding up future requests in case of a cache hit. In case of a cache miss, a cache needs to obtain the information elsewhere, which is comparatively slower. Stuedi et al. [3] describe a data cache for mobile devices (WhereStore). WhereStore replicates cloud data on the limited memory of a mobile device based on the device's location history. As a result, it decreases data access times and improves data availability, especially in case of a disconnection between a mobile device and the cloud. The main idea of this location-aware local caching system between mobile devices and the cloud is that only specific data is typically accessed at certain places (e.g., restaurant recommendations at downtown). WhereStore distributes data between mobile devices and the cloud based on filtered replication combined with user's location history. Michie [4] proposes a local run-time optimization related to function calls, which avoids the overhead of redundant local function execution (Memoization). To this end, a local function cache stores function arguments and the corresponding function's result of previous function calls. Thus, a function call is returned immediately with the result, if the local function cache has a corresponding result. This avoids the execution of the function's body once again. Both approaches utilize a local (data/function) cache to reduce data access time (WhereStore) and function execution time (Memoization), respectively. However, both approaches do not consider the integration of a cache into code offloading.

Several code offloading approaches show the general improvement of a mobile device's energy efficiency. For instance, MAUI [1] and CloneCloud [2] are code offloading systems which offload code to a remote server in the Internet. To determine the execution side of offloadable code, both approaches formulate an optimization problem minimizing the device's energy consumption subject to a maximal application execution time. MAUI and CloneCloud focus on the improvement of energy efficiency related to code offloading, however, not regarding monetary costs caused by executing code on a remote server (e.g., located in a commercial cloud).

A code offloading approach explicitly including the costs of utilizing commercial cloud services is ThinkAir by Kosta et al. [5]. ThinkAir scales the computational power of the remote server dynamically to achieve an optimal performance. To this end, ThinkAir performs on-demand resource allocation as well as exploiting parallelism. Besides ThinkAir, further code offloading approaches consider monetary costs like [6] or [7]. Ferber et al. [6] propose a middleware which augments the execution of compute-intensive applications on a mobile device with cloud resources. Based on on-demand provisioning of the server utilization in the cloud, they evaluate the arising costs by accounting the end user. Shi et al. [7] present COSMOS, which also allocates cloud resources to schedule efficiently the remote execution of offloaded code. It offers computation offloading as a service, where offloaded code from mobile devices is mapped dynamically to compute resources from a commercial cloud. As a result, the offloading optimization problem of COSMOS minimizes the leasing costs of cloud resources while handling variable network connectivity. All three approaches consider the costs of utilizing commercial cloud services, however, they do not consider the integration of a remote-side cache to code offloading.

Jiang et al. [8] explicitly consider a cache for code offloading in their approach (ADEOM). ADEOM reduces the transmission time between a mobile device and a remote server by integrating a server cache. This server cache stores objects that a remote server needs for the execution of offloaded code. In case of a repeated remote execution of previously offloaded code, mobile devices do not have to sent objects stored in the cache. As a result, ADEOM decreases the memory size of migrated data and accelerates the transmission time. However, ADEOM does not consider the optimization of a remote execution by caching the result of previously executed code.

III. System Model & Problem Statement

In the following section, we give an overview of the system model and state the problem to be solved.

A. System Model

The system model consists of four components (cf. Figure 1): A battery-operated *mobile device*, an *offloading server*, a *caching server*, and a *communication network*.

The battery-operated *mobile device* executes mobile applications written in Java. To this end, it utilizes a standard Java Virtual Machine (JVM) – the OpenJDK Java Platform in our implementation. We extended the OpenJDK JVM with our Caching-enabled Code Offloading Framework (*C2OF*). At

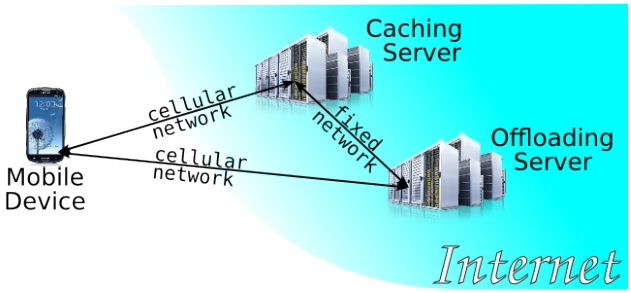

Fig. 1. Overview of system model.

run-time, *C2OF* identifies suitable application parts of a mobile application for code offloading. Moreover, *C2OF* decides whether it executes an application part locally, queries the *caching server*, or/and migrates code to the *offloading server* taking into account current system context like bandwidth. *C2OF* migrates code at the granularity of functions (more precisely, Java methods). For both cache query and code migration, *C2OF* sends the required information from the mobile device to the caching server or offloading server. The required information for a cache query consists of a unique identifier of the called function. The required information for code migration consists of the code, execution state information, and signature of the called function (*offloading request*). In both cases, the mobile device waits in idle mode until it receives a reply. A reply from a caching server can be a cache hit message or a cache miss message, where a cache hit contains the cached *offloading response*. An offloading response comprises the function's result, execution state information (e.g., changed objects), and function's signature. A reply from an offloading server contains also an offloading response, now, from the remote execution of the function.

The *offloading server* is a classic server machine, for instance, hosted in a cloud data center. It provides Offloading as a Service (OaaS) to mobile devices with a pay-as-you-go cost model. The (remote) execution time determines the monetary cost for an offloading request. To execute a migrated application part on behalf of a mobile device, the offloading server utilizes the same type of JVM with the *C2OF* as the mobile device to abstract from the actual physical hardware. After the offloading server has executed an offloading request, it sends an offloading response back to a mobile device.

The *caching server* is also a classic server machine, for instance, hosted in a cloud data center. It provides Caching as a Service (CaaS) to mobile devices with a pay-as-you-go cost model. We assume that each cache query or a cache insert cause monetary cost (cf. Google's Memcache). To handle cache queries or cache inserts, a caching server consists of a database, storing key-value pairs. A key identifies uniquely a migrated application part from an application. A value equals to the *offloading response* from a migrated application part. Mobile devices as well as offloading servers can query a caching server for key-value pairs as well as insert key-value pairs into a caching server.

The *communication network* consists of a mobile communication network (e.g., cellular LTE network) between mobile devices and the Internet. The offloading server and caching server communicate via a fixed network.

B. Problem Statement

The overall goal of our approach is to minimize energy consumption and monetary cost for application execution on a mobile device subjected to a maximum execution time. A mobile device consumes energy for (1) executing code on local hardware (*local*), (2) identifying a suitable application part, querying a caching server, and receiving an offloading response (*cache*), and/or (3) identifying a suitable application part, migrating it to an offloading server, waiting in idle mode during remote execution, and receiving an offloading response (*offload*). A mobile device causes monetary cost for (1) querying a caching server (*CaaS*) and/or (2) executing code on an offloading server (*OaaS*). Moreover, we introduce a constraint on the maximum execution time of offloaded code to improve user experience. For instance, offloading code in case of a limited network quality keeps energy consumption low but increases overall execution time. Formally, the objective for a mobile application A is to minimize a cost function u_c under a given maximum execution time constraint $T_{\max}(A)$:

$$\begin{aligned} \min \quad & u_c(E(A), M(A)) \\ s.t. \quad & T(A) \leq T_{\max}(A) \end{aligned} \tag{1}$$

where cost function $u_c(E(A), M(A))$ calculates a weighted sum of related energy consumption $E(A)$ with an energy weight w_e and related monetary cost $M(A)$ with a monetary weight w_m: $u_c(E(A), M(A)) = w_e \cdot E(A) + w_m \cdot M(A)$. $T(A)$ corresponds to the application execution time and $T_{\max}(A)$ to the maximum execution time of application A.

Because a Java application consists of successive function calls $f_i \in f$ with $i = 0, 1, \ldots, n$ (interacting with each other), we evaluate energy consumption, monetary cost, and execution time at the granularity of functions. In more detail, the energy consumption $E(A)$ for a mobile application A is the sum of (1) energy for local function execution ($E_{\text{local}}(f_i)$), (2) energy for function caching ($E_{\text{cache}}(f_i)$), and (3) energy for function offloading ($E_{\text{offload}}(f_i)$):

$$E(A) = \sum_{f_i \in f^l} E_{\text{local}}(f_i) + \sum_{f_i \in f^q} E_{\text{cache}}(f_i) + \sum_{f_i \in f^o} E_{\text{offload}}(f_i)$$

where set f^l contains all local executed functions, set f^q all functions with a cache query, and set f^o all remote executed functions ($f^l \cup f^q \cup f^o = f$).

The monetary cost $M(A)$ corresponds to fees (1) for sending queries to a caching server (CaaS) and (2) for remote executed functions on an offloading server (OaaS):

$$M(A) = \sum_{f_i \in f^q} C_{\text{cache}}(f_i) + \sum_{f_i \in f^o} C_{\text{offload}}(f_i)$$

The execution time $T(A)$ is the sum of execution times for locally executed functions, cache queries to a caching server, and remotely executed functions on an offloading server:

$$T(A) = \sum_{f_i \in f^l} T_{\text{local}}(f_i) + \sum_{f_i \in f^q} T_{\text{cache}}(f_i) + \sum_{f_i \in f^o} T_{\text{offload}}(f_i)$$

In summary, caching (identifying, querying, and receiving) improves offloading (identifying, migrating, waiting, and receiving) for a function f_i through speeding up the migrating and waiting through a cache query. Optimally, each cache query corresponds to a cache hit, where only $E_{\text{cache}}(f_i)$, $C_{\text{cache}}(f_i)$, and $T_{\text{cache}}(f_i)$ arise. Otherwise, additionally offloading cost arise: $E_{\text{offload}}(f_i)$, $C_{\text{offload}}(f_i)$, and $T_{\text{offload}}(f_i)$.

IV. Caching-enabled Code Offloading Framework

In this section, we describe our Caching-enabled Code Offloading Framework (*C2OF*). We start with an overview, before we describe in detail the integration of a remote-side cache to code offloading.

A. Overview

C2OF supports code offloading between a mobile device and an offloading server by executing an offloading-enabled JVM. This is a standard JVM extended by three further components (cf. Figure 2): an *Offloading Compiler*, an *Offloading Controller*, and an *Offloading Monitor*.

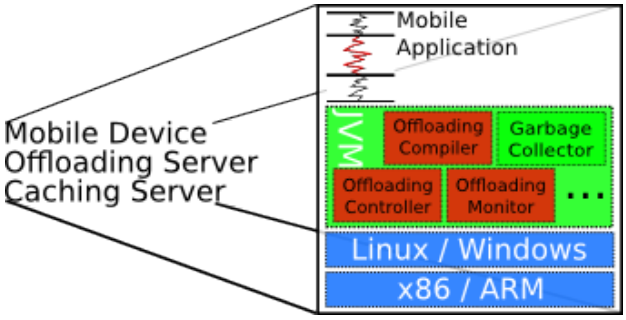

Fig. 2. Overview of system architecture.

Based on these three components, *C2OF* identifies suitable functions for code offloading. Suitable functions possess long running computations with small input and output parameters, not accessing local resources (e.g., GPS). To minimize an application's energy consumption on a mobile device, code offloading executes functions on a powerful offloading server, where the mobile device's energy consumption for migrating, waiting, and receiving is smaller than for a local execution. To execute a suitable function remotely, the offloading server requires local execution state information from a mobile device. We call this set of required information a safe-point. A safe-point comprises the function's execution state (e.g., function parameters), all referenced data (e.g., heap objects) of the function, and the function's code (Java bytecode). For an efficient safe-point generation, *C2OF* utilizes the methods described in our previous publication [9]. Together with the application's name and function's signature, a safe-point represents an offloading request, identifying a function call uniquely. When an offloading server receives an offloading request from a mobile device, it installs the contained information and starts the remote execution of the offloaded function call. As an offloading request contains all required information for this function call, no further communication occurs during the remote function execution. After the offloading server finishes the remote function execution, it creates a further safe-point containing now the function's result and all changed data. Together with the functions signature, it transmits this offloading response back to the mobile device. On receiving the offloading response, the mobile device installs the contained information and continues the application execution.

The integration of a remote-side cache to this basic code offloading improves the energy consumption and saves monetary cost. The remote-side cache is a Java application executed on a server instance in the cloud. The cache storage consists of a huge hash map, which maps unique *keys* to *values*. Everytime *C2OF* on the mobile device decides to migrate a function to an offloading server, it first sends a cache query to a caching server. To keep the cache-related overhead small, a cache query only consists of a Globally Unique Identifier (GUID) instead of the full offloading request (order of bytes vs. kilo/megabytes). To this end, *C2OF* creates three hash values from an offloading request: a hash value of the application's complete name like `app_name-v1.3`, a hash value of the offloaded function's entire signature like `v.s.Klass.call(Ljava.lang.String;J[F)S`, and a hash value of the collected safe-point. For hash computation, *C2OF* utilizes the efficient DJB hash function (cf. Henke et al. [10]). As a result, these three hash values identify a current function call uniquely. Everytime a caching server receives a GUID (*key*), it starts a cache look-up within its local storage and sends a stored offloading response (*value*) back to a mobile device, if it contains a corresponding value for the key (cache hit). In this case, *C2OF* installs the received safe-point on the mobile device, exits the called function and continues application execution. Otherwise, *C2OF* receives a cache miss reply and starts the basic code offloading process.

After the remote execution of an offloaded function, the offloading server also sends the GUID and the offloading response to a caching server. The caching server then updates the key-value storage with appropriate entries proactively. To increase the efficiency of the cache (hit ratio), we use application-specific caches. An application-specific cache only stores key-value pairs for a certain mobile application. To this end, all offloading servers send their cache inserts – caused by different mobile devices executing the same mobile application – to the same application-specific cache. Compared to a general cache that provides a storage for any application, an application-specific cache increases the cache hit ratio for a cache request, because it only contains application-related entries. If the cache storage reaches its memory limit, it starts a Last-Recently-Used cache replacement strategy.

B. Mechanisms

In the following, we explain in detail the *Offloading Compiler*, the *Offloading Controller*, and the *Offloading Monitor*.

1) Development Time: At development time, an application developer annotates functions as `Offloadable` like in [1]. An optimal offloading function uses local resources heavily (e.g., a chess AI engine) with only small input parameters, no further required input data (e.g., `static` heap objects), and a small return value. Besides small input and output values, the function must not access local resources (e.g., GPS or camera). The Java *Offloading Compiler* compiles the Java source code of an application into Java bytecode at development time. Moreover, it introduces two further Java instructions for annotated functions: `offload` and `offload_end`. The *Offloading Compiler* integrates the instruction `offload` at the start of a function's body and the instruction `offload_end` at the body's end. Based on these two instructions, the *Offloading Controller* identifies suitable functions at run-time.

2) Run-Time: At run-time, *C2OF* executes the bytecode of the Java application. Everytime it finds the bytecode `offload`, which indicates the current called function as a suitable function for code offloading, it invokes its *Offloading Controller*. The *Offloading Controller* then solves the optimization problem of Equation 1 for the currently called function. Note that $E(A)$, $M(A)$, and $T(A)$ in general depend on dynamic parameters only known at run-time. For instance, in a chess game the moves of the user determine the input parameters of the function. Therefore, the optimization problem cannot be solved offline but rather has to be solved online based on the current parameters. The *Offloading Controller* first considers for a current called function f_c the tradeoff between executing it on the mobile device or receiving a cache hit for it from a caching server (cf. Line 4 of Algorithm 1). If, for instance, the network quality is too bad (e.g., due to a high network latency), the *Offloading Controller* decides to execute the function on the mobile device (cf. Line 17). Otherwise, if the *Offloading Controller* sends a cache query and receives a cache hit, it exits the called function with the received result (cf. Line 6). If it receives a cache miss, it considers a further tradeoff between executing the function on the mobile device or offloading it to the offloading server (cf. Line 10).

Regarding the caching-enabled offloading decision-making Algorithm 1, the *Offloading Controller* requires the following three input parameters: called function's characteristic, current network conditions, and device's (energy and execution) characteristic. Function's characteristic includes input size, output size, and number of executed instructions, which the *Offloading Controller* determines with a history-based approach (cf. Kosta et al. [5]). To this end, the *Offloading Monitor* measures the number of executed instructions for suitable functions using the two offload bytecodes introduced above. It then stores the measured instruction number, input size, and output size in a local database. Furthermore, the *Offloading Monitor* monitors current network conditions (e.g., bandwidth) and provides this information also to the optimization problem. Finally, device-specific energy factors (e.g., consumed energy for receiving data) and device-specific execution factors (e.g., execution speed for Java bytecodes) determine a mobile device's (energy and execution) characteristic. These energy and execution factors are defined offline.

Algorithm 1: Decision-making algorithm for a function f_c

1: $W^u_{local} = w_e \cdot E_{local}(f_c)$
2: $W^u_{cache} = w_e \cdot (E_{query}(f_c) + E_{result}(f_c)) + w_m \cdot (M_{query}(f_c) + M_{result}(f_c))$
3: $W^t_{cache} = T_{query}(f_c) + T_{result}(f_c)$
4: **if** $W^u_{cache} < W^u_{local}$ && $W^t_{cache} < T^{f_c}_{max}$ **then**
5: **if** `SEND_CACHE_QUERY()` **then**
6: **return** `RECEIVED_RESULT()`
7: **else**
8: $W^u_{offload} = w_e \cdot (E_{miwa}(f_c) + E_{result}(f_c)) + w_m \cdot (M_{miwa}(f_c) + M_{result}(f_c))$
9: $W^t_{offload} = T_{miwa}(f_c) + T_{result}(f_c)$
10: **if** $W^u_{offload} < W^u_{local}$ && $W^t_{offload} < T^{f_c}_{max}$ **then**
11: **if** `SEND_OFFLOAD_REQUEST()` **then**
12: **return** `RECEIVED_RESULT()`
13: **end if**
14: **end if**
15: **end if**
16: **end if**
17: `EXECUTE_LOCAL()`

Based on these input parameters, energy consumption E for a function call f_c is defined as:

$$
\begin{aligned}
E_{\text{local}}(f_c) &= e_{\text{local}} \cdot \frac{f_c^{\text{instr}}}{s_{\text{local}}} \\
E_{\text{result}}(f_c) &= e_{\text{receive}} \cdot \frac{f_c^{\text{output}}}{b_{\text{down}}} \\
E_{\text{query}}(f_c) &= e_{\text{send}} \cdot \frac{f_c^{\text{hash}}}{b_{\text{up}}} + e_{\text{idle}} \cdot t^{Q}_{\text{cache}} \\
E_{\text{miwa}}(f_c) &= e_{\text{send}} \cdot \frac{f_c^{\text{input}}}{b_{\text{up}}} + e_{\text{idle}} \cdot \frac{f_c^{\text{instr}}}{s_{\text{remote}}}
\end{aligned}
$$

where e_{local} denotes the energy factor for executing code on local compute resource, e_{idle} for waiting in idle mode, e_{receive} for receiving data, and e_{send} for sending data. f_c^{instr} denotes the total number of executed instructions for function f_c and f_c^{hash} the size of function's GUID. f_c^{input}/f_c^{output} denote the size of the offloading request/response and b_{up}/b_{down} the uplink/downlink network bandwidth. s_{local} (s_{remote}) denotes the local (remote) instruction processing speed. Finally, $t^{Q|I}_{\text{cache}}$ denotes the duration of a cache *Q*uery/*I*nsert.

The monetary cost M for a method f_c is pay-per-use, where a mobile end user pays for the on-demand provisioned cloud services (OaaS and CaaS):

$$
\begin{aligned}
M_{\text{result}}(f_c) &= c_{\text{send}} \cdot f_c^{\text{output}} \\
M_{\text{query}}(f_c) &= c_{\text{receive}} \cdot f_c^{\text{hash}} + c_{\text{cache}} \cdot t^{Q}_{\text{cache}} \\
M_{\text{miwa}}(f_c) &= c_{\text{receive}} \cdot f_c^{\text{input}} + c_{\text{remote}} \cdot \frac{f_c^{\text{instr}}}{s_{\text{remote}}} + c_{\text{cache}} \cdot t^{I}_{\text{cache}}
\end{aligned}
$$

where c_{send} (c_{receive}) equals to the costs of sending (receiving) data. c_{remote} defines the cost of executing code on a remote server (OaaS) and c_{cache} defines the caching costs (CaaS).

The method execution time T is defined as:

$$
\begin{aligned}
T_{\text{local}}(f_c) &= \frac{f_c^{\text{instr}}}{s_{\text{local}}} \\
T_{\text{result}}(f_c) &= \frac{f_c^{\text{output}}}{b_{\text{down}}} \\
T_{\text{query}}(f_c) &= \frac{f_c^{\text{hash}}}{b_{\text{up}}} + t_{\text{cache}}^{Q} \\
T_{\text{miwa}}(f_c) &= \frac{f_c^{\text{input}}}{b_{\text{up}}} + \frac{f_c^{\text{instr}}}{s_{\text{remote}}}
\end{aligned}
$$

In summary, set f^l contains all functions that the application developer has not annotated as `Offloadable` and all annotated functions with a local execution (cf. Line 17 of Algorithm 1), for instance, due to a too bad network quality. Set f^q contains all functions with a cache query (cf. Line 5) and f^o with an offload request (cf. Line 11).

V. Evaluation

Next, we evaluate energy efficiency and monetary cost savings of our Caching-enabled Code Offloading Framework (*C2OF*). We describe our evaluation setup and present the results.

A. Setup of Experiments

Since the performance of our *C2OF* depends on multiple parameters, we measured execution time, energy consumption, and monetary cost of two different mobile devices in real environments, executing two different types of mobile applications.

For the mobile applications, we chose a *chess game* and a *speech synthesizer*. Both differ significantly w.r.t. computational complexity and communication overhead and therefore benefit differently from offloading and caching. The *chess game* has an ideal function for code offloading, which corresponds to the calculation of the next best chess move of a computer opponent. This function has a high computational complexity and a small parameter and result size. Due to the small sizes, this function is also a good caching candidate. Regarding the huge number of possible chess positions, however, the cache hit ratio decreases fast with the number of played chess rounds. Therefore, we evaluated the *chess game* with different start parameters, where *R* equals to the number of chess rounds and *S* to the selected start piece. The *speech synthesizer* has also a good function for code offloading, which corresponds to the conversion from words into speech. This function has small input parameters but a big result (e.g., audio files). Compared to the *chess game*, the cloud-related speed-up is smaller for the *speech synthesizer*. However, *speech synthesizer* is a better caching candidate, because the cache hit ratio depends only on the general word frequency within a text. Therefore, we evaluated the *speech synthesizer* with different start parameters, where *R* equals to the total number of read words, *S* to the start index, and *F* to the selected text.

For the mobile devices, we chose two heterogeneous classes: 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 two GByte of RAM. Both mobile devices communicate via WiFi with an offloading server and a caching server. For the offloading server and caching server, we use a standard PC server with an Intel Core i7-2600 Quad-Core processor (3.4 GHz) and eight GByte of RAM. All three devices run a Linux as operating system. We consider the cloud computing prices from Google and Amazon. Since these providers do not charge for communication, c_{send} and c_{receive} are set to zero (cf. Algorithm 1). For c_{cache} and c_{remote} we consider the prices for a *m3.medium* machine type from Amazon (causing monetary costs of 0.070 \$ per hour) as well as an *n1-standard-1* machine type from Google (causing monetary costs of 0.063 \$ per hour). Both OaaS or CaaS are executed on an instance in the cloud, why we assume twice the average for an OaaS or CaaS, resulting in $3.69 \cdot 10^{-5}$ $\left(= \frac{0.070+0.063}{60 \cdot 60}\right)$ \$ per second. Thus, an OaaS provider, utilizing a Google *n1-standard-1* instance, earns money after 1705.26 s of code executions on its cloud instance for multiple offloading requests.

For the number of cache entries, we compare our caching-enabled code offloading approach in different configurations (*Cached*$_{Empty}$, *Cached*$_{Full}$, and *Cached*) with *Local* and *Baseline*. *Local* executes the mobile applications on an unaltered JVM (without offloading or caching). *Baseline* corresponds to a basic code offloading approach. *Cached*$_{Empty}$ (*Cached*$_{Full}$) represents the caching worst case (caching optimal case), where the cache does not have any entries (has all queried entries) and thus, answers every time with a cache miss (cache hit). Beside worst case and best case evaluation, we further evaluated both mobile applications together in a *mobile scenario*, providing some insights about the actual performance. To this end, each mobile device executes simultaneously a *chess game* and/or a *speech synthesizer* multiple times with different parameters. Both utilize the same application-specific cache, which is empty at the start of the mobile scenario (*Cached*). To keep the results of *Baseline* comparable with *Cached*, we only utilized two mobile devices, because the cache hit ratio increases with the number of participating nodes. We limited the total number of cache memory to 1 GB, which was large enough to keep all stored results in memory.

B. Results: Mobile Chess Game

First, we compare the evaluation results of the *chess game*, wherefore Figure 3 shows the execution times for the netbook. The blue bars correspond to *Local* which possesses the longest execution times (maximum: 345.03 s; minimum: 134.70 s). Compared to *Baseline* (cf. orange bars) with a maximum value of 45.31 s and a minimal value of 24.44 s, code offloading reduces the execution time by up to 86% and energy consumption by up to 88%. Regarding the execution times of *Cached*$_{Empty}$, our approach keeps the cache-related (computation and communication) overhead compared to *Baseline* very small (cf. grey and orange bars). In detail, the difference between the execution times are in the order of ten milliseconds owing to the additional cache queries. The

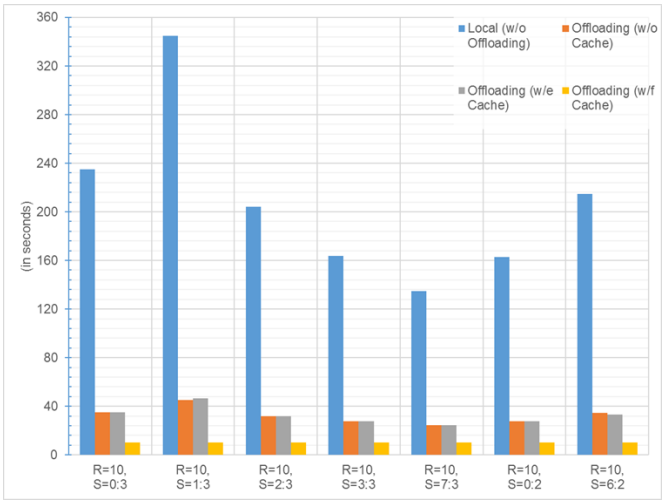

Fig. 3. Netbook: Execution times of *chess game*, where *R* equals to the number of chess rounds and *S* to the selected start piece.

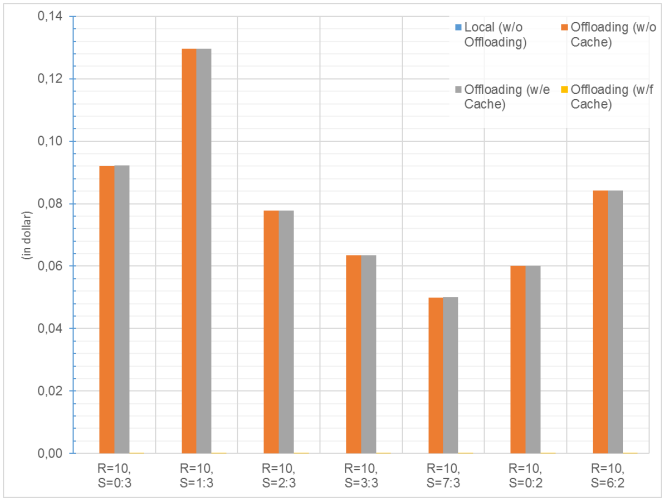

Fig. 4. Laptop: Monetary costs of *chess game*, where *R* equals to the number of chess rounds and *S* to the selected start piece.

execution time reduction of $Cached_{Full}$ compared to *Baseline* are significant, reducing time by up to 77% and energy by up to 74%. Overall, $Cached_{Full}$ reduces the execution time and the energy consumption compared to *Local* by up to 97%.

For the laptop, the same time and energy characteristics apply as for the netbook. The time reduction and energy saving of *Baseline* compared to *Local* stays nearly the same, resulting in a time reduction of up to 74% and an energy saving of up to 81%. Comparing *Baseline* with $Cached_{Empty}$ and $Cached_{Full}$, the introduced overhead for caching is also very small, while the time reduction and energy saving are very high (time reduction: up to 78%; energy saving: up to 70%). Overall, for the laptop, $Cached_{Full}$ reduces the execution time and the energy consumption compared to *Local* by up to 94%.

As the characteristics of the monetary costs for the *chess game* are equal on the netbook and on the laptop, Figure 4 depicts only the monetary costs for the laptop. On both

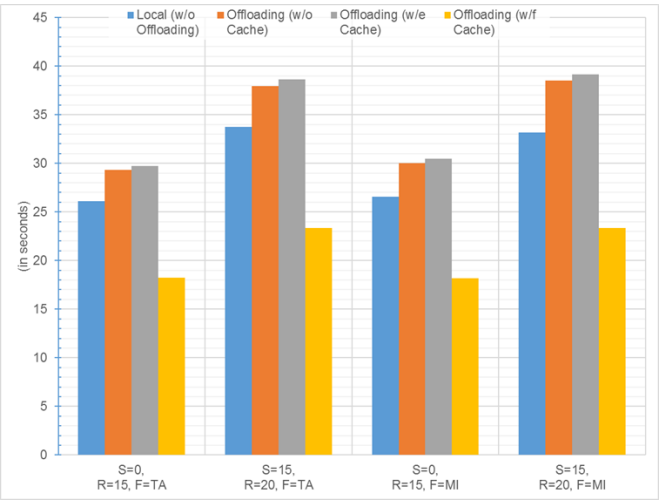

Fig. 5. Laptop: Execution times of *speech synthesizer*, where *R* equals to the total number of read words, *S* to the start index, and *F* to the selected text.

devices, *Local* causes no costs at all (cf. blue bars). The costs for *Baseline* (cf. orange bars) and $Cached_{Empty}$ (cf. grey bars) are similar, because the cost for a cache query compared to the cost for a remote execution is very small. The maximal difference of *Baseline* and $Cached_{Empty}$ is $4.02 \cdot 10^{-5}$ \$ on the netbook and $4.79 \cdot 10^{-5}$ \$ on the laptop. For $Cached_{Full}$ (cf. yellow bars) the monetary costs tend to zero causing costs between $2.4 \cdot 10^{-5}$ \$ and $4.8 \cdot 10^{-5}$ \$ on both devices.

Summarizing, our caching-enabled code offloading approach reduces for both mobile devices the energy consumption, monetary cost, and execution time of the *chess game* significantly. Because it sends only a GUID to a caching server, the number of transferred bytes is lowered significantly compared to sending a complete offloading request to an offloading server (order of bytes vs. multiple kilobytes). As a result, the (additional) cache-related overhead is very small.

C. Results: Speech Synthesizer

For the *speech synthesizer*, we also evaluated the performance on both mobile devices with different start parameters. Regarding the execution time for the netbook, *Local* takes the longest time (maximum: 85.024 s; minimum: 65.532 s). Because *speech synthesizer* is also on the netbook a good offloading candidate, *Baseline* reduces the execution time by up to 51% and energy consumption by 54%. The difference between *Baseline* and $Cached_{Empty}$ is only minor (in the order of milliseconds), highlighting the low cache-related overhead. $Cached_{Full}$ results in the lowest execution time, reducing *Baseline* by up to 35% and *Local* by up to 68%, respectively and energy consumption by 31% and 68%, respectively.

For the laptop, the *speech synthesizer* is not a good offloading candidate due to the more powerful capabilities of the laptop and the bigger communication overhead of the audio files. As a result, *Baseline* would not offload any method and, thus, also $Cached_{Empty}$ or $Cached_{Full}$ which, however, can still send cache queries. For a better comparison, the orange bars

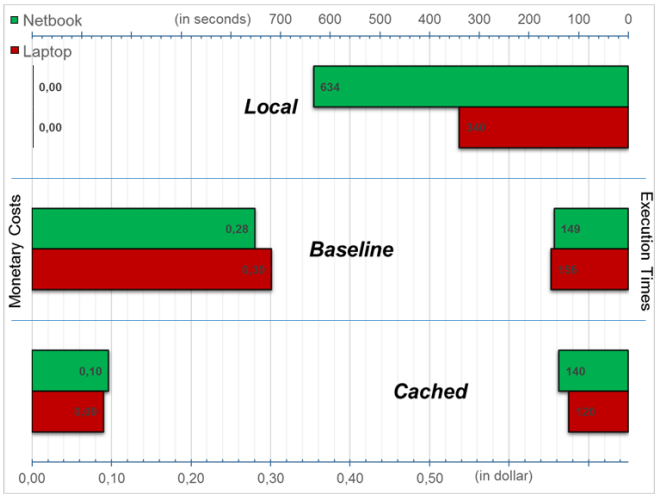

Fig. 6. Execution times and monetary costs of the *mobile scenario*.

in Figure 5 corresponds to a naive *Baseline* approach that still offloads methods, resulting in an execution time increase of about 16% and an energy increase of about 10% compared to *Local*. A naive $Cached_{Empty}$ approach also results in an execution time and energy increase (cf. grey bars). $Cached_{Full}$, however, decreases the execution time and energy consumption by up to 31% compared to *Local*. For the monetary costs, *Local* causes no costs, whereas a naive *Baseline* approach causes monetary costs between $3.80 \cdot 10^{-2}$ \$ and $5.56 \cdot 10^{-2}$ \$ for both netbook and laptop. The monetary costs of $Cached_{Full}$ are between $4.17 \cdot 10^{-5}$ \$ and $1.53 \cdot 10^{-4}$ \$.

Summarizing, our caching-enabled code offloading approach performs also for the *speech synthesizer* very well, despite the fact that the *speech synthesizer* would not benefit from basic code offloading on the laptop.

D. Results: Mobile Scenario

Figure 6 shows the evaluation results for *Local*, *Baseline*, and *Cached* for the mobile scenario on the netbook (cf. green bars) and on the laptop (cf. red bars). Comparing the execution times of all three approaches (cf. green/red bars from right to left), *Local* takes the longest execution time (netbook: 633.52 s; laptop: 340.42 s). *Baseline* and *Cached* result in similar execution times, because the cache is empty at the start, filled up during the *mobile scenario* from both mobile devices. *Baseline* reduces *Local*'s execution time by up to 76% on the netbook (laptop: up to 54%) and *Cached* reduces *Local*'s execution time by up to 77% on the netbook (laptop: up to 64%). Regarding the monetary costs of the *mobile scenario* (cf. green/red bars from left to right), *Local* causes no monetary costs on both mobile devices, whereas *Cached* reduces the monetary costs by up to 70% due to cache hits compared to *Baseline*. As a result, our caching-enabled code offloading approach performs on both mobile devices better than basic code offloading w.r.t. energy consumption and execution time. Moreover, it reduces the caused monetary costs significantly compared to basic code offloading.

VI. Conclusion

Offloading resource-intensive application parts from a mobile application at run-time to a remote server promises to increase the energy-efficiency of a resource-constrained mobile device. Although several approaches show the general effectiveness of code offloading, they largely neglect possible remote-side optimizations like a function cache.

In this paper, we presented a novel code offloading approach which integrates a remote-side cache into code offloading, storing results of offloaded functions from multiple mobile devices. Thereupon, a mobile device queries the remote-side cache for a result of an offloadable function to avoid, on the one hand, a repeated execution of previously executed application parts on remote servers, and reduces, on the other hand, the required *remote* time in case of a cache hit. As a result, a mobile device benefits not only from shorter execution times of offloaded application parts but also from money savings by avoiding the utilization of a remote server in case of a cache hit. Our evaluation results show that this approach increases energy efficiency and decreases monetary cost significantly compared to basic code offloading approaches.

In future work, we want to further improve the cache hit ratio of the remote-side cache. For instance, a hierarchical cache structure could improve the cache hit ratio by sending important cache entries up the hierarchy. Another open research question is to consider privacy implications of such a remote-side cache for code offloading. For instance, an attacker could derive information (e.g., the execution flow) from another end user that utilizes a remote-side cache.

References

[1] 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*, ser. MobiSys'10, March 2010, pp. 49–62.

[2] 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*, ser. EuroSys'11, 2011, pp. 301–314.

[3] P. Stuedi, I. Mohomed, and D. Terry, "WhereStore: Location-based Data Storage for Mobile Devices Interacting with the Cloud," in *Proc. 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, 2010, pp. 1:1–1:8.

[4] D. Michie, "Memo Functions and Machine Learning," *Nature*, vol. 218, no. 5136, pp. 19–22, April 1968.

[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*, March 2012, pp. 945–953.

[6] M. Ferber, T. Rauber, M. H. C. Torres, and T. Holvoet, "Resource Allocation for Cloud-Assisted Mobile Applications," in *Proc. IEEE 5th Intl. Conf. Cloud Computing*, June 2012, pp. 400–407.

[7] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS: Computation Offloading As a Service for Mobile Devices," in *Proc. 15th ACM Intl. Symposium on Mobile Ad Hoc Networking and Computing*, August 2014, pp. 287–296.

[8] Y. Jiang, J. He, Q. Li, and X. Xiao, "A Dynamic Execution Offloading Model for Efficient Mobile Cloud Computing," in *Global Communications Conference, 2014 IEEE*, December 2014, pp. 2302–2307.

[9] 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*, June 2014, pp. 76–83.

[10] C. Henke, C. Schmoll, and T. Zseby, "Empirical Evaluation of Hash Functions for Multipoint Measurements," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 39–50, July 2008.