Increasing the Efficiency and Responsiveness of Mobile Applications with Preemptable Code Offloading

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

Abstract—Mobile applications are getting more and more sophisticated and demanding. Although the processing, memory, and storage resources of mobile devices are constantly increasing to enable such resource-demanding mobile applications, battery capacity is still the main limiting factor. To solve this problem, mobile code offloading approaches can be used to offload parts of a mobile application to remote servers and utilize the resources of cloud services.

In this paper, we propose a novel code offloading approach that makes code offloading robust against communication link failures, which are still a major problem of mobile systems. To this end, we propose preemptable code offloading. It allows for interrupting the offloading process and continuing the remote execution locally after a link failure, without abandoning the complete result calculated remotely so far. The basic idea of our approach is to create safe-points of the remote execution and transmit these intermediate results back to the mobile device. After a link failure, the mobile device can now continue execution from the last transmitted safe-point. Although safe-points induce communication and energy overhead, our evaluations show that using an optimized safe-point schedule this overhead quickly pays off under link failures. Besides reducing the overall energy consumption significantly, responsiveness also benefits from safe-points by meeting given execution deadlines after link failures.

Keywords—Distributed Systems, Code Offloading, Safe-points, Mobile Cloud Computing, Efficiency, Responsiveness

I. INTRODUCTION

Mobile applications are evolving more and more into so-called Rich Mobile Applications with sophisticated functionality and increasing resource demands. Although the computational, memory, and storage resources of modern smart phones or tablets are improving rapidly, battery capacities still increase quite moderately. Therefore, energy-efficiency remains one of the greatest challenges in mobile computing. To improve the energy-efficiency, mobile applications can be assisted through a cloud infrastructure by dynamically off-loading functionality from the resource-constrained mobile device to a virtually unlimited server in the cloud. Several approaches for mobile code offloading such as CloneCloud [1] or ThinkAir [2] have been proposed in the literature. These approaches show the general effectiveness and feasibility of mobile code offloading in well-connected mobile environments. Owing to being (almost) always online, they neglect the difficulties and problems, arising from connection failures. As evaluations in the literature shows, mobile connections are still far from faultless, resulting in situations where the mobile device is (temporary) offline. For instance, Ding et al. [3] collected a dataset from 3785 volunteers worldwide, measuring 3G signal strength of their Android smart phones. Thereupon, they regard that over 80% (60%) of the users have a poor signal strength for over 15% (32%) of their active usage time. Since smart phone users move frequently in an area with poor cellular network coverage, they identified the geographic variations of network coverage as the major reason for temporary disconnections. However, current offloading approaches consider only very limited error strategies to deal with such temporary link failures. They either re-execute totally the migrated code locally on the mobile device or wait for a re-connection to the cloud infrastructure. Both strategies have adverse effects on the energy efficiency: On the one hand, offloading plus local re-execution requires more energy than only performing the code locally, since the device spends some idle time waiting for the result of the migrated code. Usually, the display is not turned off during this time, therefore the spent energy is not negligible. If, now, the mobile device aborts the waiting due to an error, the energy spent during this idle time as well as for the offloading decision-making is wasted. On the other hand, offloading plus waiting for the return of the remote site might block the mobile device until a re-connection, which can be arbitrary long. Moreover, it again spends energy while waiting, possibly beyond the point, where a local execution would have been more efficient.

In order to solve this dilemma of waiting for the final result or starting a local re-execution after a communication failure, we propose a novel approach for code offloading in mobile environments with disconnectivities. The basic idea of our approach is to enable the re-use of partial results. At the remote side, these are captured simultaneously to the execution of the migrated code and transferred back to the mobile device. As a result, the mobile device has no need to wait for the final result, which might not be available due to a disconnection. To this end, we create safe-points of the remote execution state and transmit them to the mobile
device. If the connection breaks, the mobile device can now continue from the last safe-point. The challenge in this scheme is to find the right times of creating and transmitting safe-points such that the energy-efficiency is increased. If too many safe-points are utilized, the communication overhead and energy consumption increases. If too few are generated, the risk of falling back to a very old state—before local re-execution—increases ultimately. At worst, it converges into the basic scheme without safe-points and waiting for the final result only. Moreover, the decision for creating a safe-point is influenced by the complexity of the offloaded function and the size of the transmitted state. If local re-execution is inexpensive w.r.t. energy, fewer safe-points are sufficient. Therefore, our approach estimates the energy saved by a safe-point based on online measurements to calculate the right time for safe-points such that the total energy consumption of the mobile device is effectively reduced. In summary, we make the following contributions: (1) a code offloading framework with safe-points for Java-based applications; (2) an adaptive algorithm with online measurements for scheduling safe-points, minimizing energy consumption and increasing robustness to temporary disconnections; (3) an implementation of the mechanisms on an application-layer virtual machine; and (4) an evaluation with state of the art mobile devices and application, showing the efficiency of a preemptable code offloading approach.

The rest of the paper is organized as follows. The following section describes the system model and states related problems of our approach. Afterwards, Section III gives an overview and describes required components. Then, we outline the experimental results in Section IV and characterize related work in Section V. Finally, we conclude our work and give an outlook on future work.

II. SYSTEM MODEL AND PROBLEM STATEMENT
Before we present our technical contributions, we introduce our system model and describe the problem to be solved.

A. System Model
Our system consists of four different components: mobile device, offloading server, code server, and communication network:

The mobile device is battery-operated and runs non-offloadable application code like GUI or sensor readings as well as offloadable application code that is not yet migrated to the cloud infrastructure. We assume that applications are implemented in Java, where executable code is represented by a platform-independent byte-code representation and interpreted through a Java Virtual Machine (JVM). This virtualization allows the execution of code on different hardware platforms and enables the migration of code at runtime between a mobile device and the cloud infrastructure.

The cloud infrastructure consists of offloading servers and code servers, both hosted in data centers. Offloading servers host JVMs for executing offloaded application parts. Code servers store application code and can be used by mobile devices as well as offloading servers to download application byte code. In order to relieve the mobile device from sending code over mobile communication links, the offloading server downloads required application byte code from the code servers. For the sake of simplicity, we only consider one dedicated offloading server and one dedicated code server per application with sufficient capacities, since server load balancing and horizontal scaling of the cloud infrastructure are out of scope of this paper.

For communication, mobile devices use mobile communication networks such as a 3G or 4G cellular network. Due to mobility, mobile devices might suffer from transient network outages, which are unpredictable and might last for longer periods (seconds to minutes depending on the mobility and environment of the device). Connections between infrastructure servers use the fixed Internet infrastructure, which is assumed to be reliable and high-bandwidth.

B. Problem Statement
The general goal of our offloading approach is to minimize the energy consumption of the mobile device as well as increase the application robustness to temporary disconnections. This includes two aspects: On the one hand, the energy consumption should stay small in cases with temporary disconnections. On the other hand, robustness includes soft real-time constraints that also have to be concerned when a disconnection occurs. More formally, the mobile device’s total energy budget for offloading a method can be expressed by the following term:

$$E(\text{method}) = E_{\text{local}} + E_{\text{offload}} + E_{\text{safe-points}}$$

$E_{\text{local}}$ denotes the energy spent for local processing, storage, display, etc. on the mobile device. $E_{\text{offload}}$ defines the energy overhead for offloading the method to the offloading server. This part includes the protocol overhead as well as the state transfer for the offloaded method. Note that the executable code is directly downloaded by the offloading server from the code server, so this part is not included in $E_{\text{offload}}$. $E_{\text{safe-points}}$ denotes the energy for transferring safe-points from the offloading server to the mobile device,
again including protocol overhead and state transfer. Note that the total number of transferred safe-points as well as the energy consumption of each safe-point are unknown in advance, and thus, $E_{\text{safe-points}}$.

Moreover, let $T(\text{method})$ be the total time of executing the method and $T_{\text{max}}(\text{method})$ be the maximum tolerated time for this method (time constraint). Then, we search for a safe-point strategy selecting optimal times and numbers for safe-points such that the total energy is minimal under the given time constraint:

$$\min E(\text{method})$$
$$s.t. \quad T(\text{method}) \leq T_{\text{max}}(\text{method})$$

Theoretically, it would be desirable that offloading is always more efficient than a purely local execution without offloading, i.e., $E(\text{method}) \leq E_{\text{pure-local-exec}}(\text{method})$. However, considering our assumption that network disconnections may appear suddenly without warning, we cannot always guarantee this property, but have to cope with a certain level of uncertainty. For instance, if we decided to offload a method and suffer from a disconnection immediately afterwards, the overhead does not payoff, since no safe-point exists yet from which we could profit.

### III. Preemptable Code Offloading

In this section, we present our preemptable code offloading approach. We start with an overview of the basic concept and architecture, before we describe in detail the different functionalities and involved components.

#### A. Overview

Figure 2 shows a typical execution of a mobile application using preemptable code offloading. Before the first execution of an application, the mobile device downloads the application code (Java bytecode) from a code server ($t_0$) once. When the application is started ($t_A$), the mobile device initializes a remote execution environment by sending a prepare request to the offloading server to allocate a remote JVM that can be used for offloading. This request contains the application name and version. Using this information, the offloading server prefetches the application code from a code server and acknowledges the request ($t_B$).

Typically, the execution of Java bytecode requires a compiler to translate the bytecode to machine code during runtime. To facilitate code offloading, we replace this standard compiler of the JVM by a custom client-side offloading compiler. The offloading compiler instruments the code with breakpoints such that the execution is interrupted before certain methods of the application code are called as detailed below. At these breakpoints, the client-side offloading controller decides whether the method should be offloaded to the offloading server using the offloading criteria described below. If the decision is made to offload a method from the mobile device, a safe-point generator on the mobile device is invoked to generate a local safe-point and an offloading request is sent to the offloading server ($t_C$). The transmitted safe-point includes the method parameters as well as all reachable heap objects, declared class member objects, and static objects. Then, the local execution is paused.

On receiving an offloading request ($t_D$), the offloading server starts the server-side execution of the prefetched code at the first safe-point. Again, the code is instrumented with breakpoints, this time using a server-side offloading compiler, to interrupt the execution at certain points as detailed below. At these breakpoints, the server-side offloading controller decides whether to create safe-points and send them back to the mobile device ($t_E$). During the remote execution, the server might create several safe-points. In particular, safe-points are only sent if local energy consumption since the last transferred safe-point is higher than transmitting a safe-point (cf. Section III-C).

While code is executed remotely, the client-side offloading controller runs a failure detector to detect connection failures. In case of an error, it informs the offloading controller to preempt offloading and continue execution locally at the last safe-point received from the offloading server ($t_{G'}$).

#### B. Offloading Compiler

As briefly mentioned, the offloading compilers instrument the original Java bytecode prior to execution. To this end, we extended a standard Java compiler to add offloading-specific functions to the bytecode before translating it to machine code. Both client-side and server-side compilers
insert breakpoints, however, with different functionalities: On the client-side, breakpoints are inserted before methods which are feasible candidates for offloading. To decide which methods are feasible candidates, we use similar criteria as in state-of-the-art offloading approaches like [2] or [4]. For instance, we do not offload methods accessing resources which are only available locally like sensors or HCI components. On the server-side, breakpoints are inserted before all branching points and method exit points of the Java bytecode. Branching points are if, switch, try-catch, and loop statements. They are particularly well-suited for small memory footprints of safe-points since typically variables with a local scope—for instance, inside the body of a loop—are not visible and, therefore, not included in a safe-point. Method exit points are return or exception throw statements, indicating the end of remote execution and, therefore, requiring a last safe-point.

Besides adding breakpoints, the server-side offloading compiler inserts further instructions to keep track of object modifications. Object modifications are crucial to decide, which object data has been modified since the last safe-point, keeping the transmitted state size minimal. For tracking modifications, instructions are inserted at each Java bytecode operation that changes values of an object like istore. These instructions flip a specific Bit in the Java header of the corresponding object to flag that a part of the object was modified. Beside this enhancement, further instructions are inserted to count exactly the number of executed Assembler instructions for the original bytecode since the last safe-point. To this end, the offloading compiler adds instructions at each branching point listed above to sum up the total number of executed Assembler code. Thereupon, the server-side offloading controller estimates the required execution time on the mobile device as described next to determine the optimal time for sending safe-points to the mobile device.

C. Offloading Controller

The client-side offloading controller decides whether to offload a method whenever a local breakpoint is reached. Since in this paper we focus on the preemption feature and safe-pointing rather than basic offloading, offloading decisions are made similarly to the approach described in [2], i.e., we implemented a history-based approach using execution time measurements of past executions and the current measured network conditions. The server-side offloading controller determines at each branching point whether to generate and send a safe-point. In order to evaluate a safe-point in terms of energy savings, the controller estimates whether the local execution of the code on the mobile device since the last safe-point would consume more energy than sending a corresponding safe-point now. A safe-point is only sent if it saves energy on the mobile device. The basic idea is to estimate the local execution time of the past code on the mobile device since the last safe-point. Together with a model of how much energy the device spends on average while being active for a certain period, the controller can estimate the energy consumption of a partial local execution. Additionally, the controller estimates the energy which would be spent for sending the safe-point by considering its size and a model of how much energy is required to receive a certain amount of data.

In more detail, estimating the energy of local execution includes the following parameters: (1) the number of Assembler instructions \( n_{asm} \) that were executed since the last safe-point to reach the current state. The controller gets this number from the instruction counters inserted by the offloading compiler. (2) Time \( t_{instr} \) in seconds it takes on average to execute an instruction. The client-side offloading controller learns this time from previous executions and sends the resulting value to the server-side offloading controller with the offloading request. (3) The energy \( e_{scnd} \) spent on average per time period \( \Delta t \) while waiting. This value is also learned from previous executions and transmitted to the server-side offloading controller with the offloading request.

The energy for receiving a safe-point includes the following parameters: (1) Size \( s_{sp} \) of the safe-point in bytes. In order to calculate \( s_{sp} \), the safe-point generator creates a safe-point (which might not be sent if it will not be beneficial for the mobile device). (2) Energy \( e_{byte} \) for receiving one byte and (3) time \( t_{byte} \) to transmit one byte under the current network connection.

Based on these parameters, the energy \( e_{local} \) required for local execution and \( e_{recv} \) required for receiving a safe-point can be estimated by the following terms:

\[
\begin{align*}
    e_{local} &= \frac{n_{asm}}{t_{instr}} \times e_{scnd} \\
    e_{recv} &= s_{sp} \times e_{byte} + \frac{s_{sp}}{t_{byte}} \times e_{scnd} \\
\end{align*}
\]  

(3)

The safe-point is sent, if the following condition is fulfilled:

\[
    e_{local} > e_{recv} 
\]  

(4)

Thereupon, the offloading controller ensures that, if a safe-point is sent and received by the mobile device, the energy consumption \( E(method) \) until that time is lower than \( E_{pure-local-exec(method)} \) (cf. Subsection II-B). If the execution reaches a breakpoint due to a method exit point (return or exception statement), a last safe-point is always generated and returned to the mobile device. This signals the end of the offloading phase.

Summarizing, the following algorithm illustrates the safe-point decision-making, executed by the server-side offloading controller. Owing to the server-side efficiency, the offloading controller only generates a new safe-point, if \( \frac{n_{asm}}{t_{instr}} \) is greater than an introduced time threshold \( t_{thres} \) (cf. Line 3 of Algorithm 1). Furthermore, it only sends the related safe-point, if the sum of the receiving energy \( e_{recv} \) plus a further energy threshold \( e_{thres} \) is smaller than the local required energy \( e_{local} \) (cf. Line 8 of Algorithm 1):
### Algorithm 1 Safe-point Decision-making Algorithm

**Constant:** `BRANCHING`, `RETURNING`

**Constant:** `t_instr`, `byte`, `byte`, and `scnd`

**Constant:** `t_thres_init`, `t_thres_step`, and `e_thres`

**Input:** `naxm`, `t_thres`, and `btcnx`

1. **if** `btcnx == BRANCHING** then
   2. `t_ti := naxm / t_instr``
   3. **if** `t_ti > t_thres then
      4. `t_thres := t_thres + t_thres_step``
      5. `sp := generateSafePoint();``
      6. `e_recv := sizeof(sp) * (e_byte + e_scnd);``
      7. `e_local := t_vi * e_scnd;``
      8. **if** `e_local > (e_recv + e_thres) then
          9. `sendSafePoint(sp);``
          10. `t_thres := t_thres_init;``
      11. **end if**
      12. **end if**
   **else if** `btcnx == RETURNING** then
      14. `sp := generateSafePoint();``
      15. `sendSafePoint(sp);``
   16. **end if**

---

**D. Safe-Point Generator**

Safe-point generators on the mobile device and offloading server are responsible for an efficient safe-point creation. Both client- and server-side generators are based on the same concepts. Therefore, we do not distinguish between client- and server-side safe-point generation in the following. To make safe-point generation transparent to the application programmer, we implemented our own object serializer, which does not require every class to implement individual serialization functionality (Java `Serializable` interface).

We achieve this by implementing a memory module for the JVM, which transforms Java `Primitives`, `Classes`, and `Arrays` into a platform-independent representation and vice versa. In order to minimize safe-point size, the safe-point generator only serializes required data objects and objects modified since the last transmitted safe-point (incremental safe-pointing). Modified data objects can be efficiently identified by visiting all data objects that are reachable from the offloaded method and inspecting their modification flag. The set of reachable objects includes all visible `Static`s, members of the method declaring `Class`, all local `Stack` objects, and all accessible `Heap` objects. As already introduced above, the offloading compiler inserts code to toggle the modification flag whenever a data object is changed. One important prerequisite is that only pure Java code must be used since modifications outside the JVM cannot be detected by this technique since native code cannot be easily modified by the offloading compiler.

In order to install a safe-point on the client-side, the safe-point generator combines all incremental safe-points into a single safe-point containing the latest state of the data objects. Then, our memory module replaces all values with the ones contained in the safe-point.

**E. Failure Detector**

During the remote execution, a client-side failure detector detects network partitioning due to connection problems. To this end, periodic heartbeat messages are sent from the offloading server to the mobile device (pinging). If after a timeout period no heartbeats have been received, the failure detector informs the offloading controller, which resumes immediately the local execution from the last safe-point.

---

**IV. Evaluation**

In this section, we evaluate the energy-efficiency of our preemptable code offloading approach using experiments with mobile devices. We first introduce our setup, before we present the measurements.

**A. Setup of Experiments**

In order to evaluate the energy efficiency under realistic conditions, we implemented a prototype of our approach and performed energy measurements with two heterogeneous mobile devices: a netbook representing a resource-poor and energy-efficient device, and a laptop as a more powerful mobile device with higher energy consumption. The netbook is a Dell Inspiron Mini 10v Netbook with an Intel Atom N270 processor (1.6 GHz) and 1 GByte of RAM. The laptop is a Lenovo ThinkPad T61 with an Intel Core 2 Duo T7300 (2.0 GHz) and 3 GByte of RAM. Both devices use a 3G mobile network for wireless communication (HUAWEI HSPA USB Stick; Model: E1750). To allow for a direct comparison, both devices run the same operating system, namely, a minimal Linux system (Debian Wheezy). The offloading server is a standard PC (Intel Core i7-2600 Quad-Core CPU—running at 3.4 GHz—and 8 GByte of RAM) with a fixed 1 Gbps Internet connection.

We implemented the components described in Section III for the Jikes Research Virtual Machine (jRVM). As application, we use a chess game (jCGA) based on the Java chess library Chesspresso. jCGA is a typical compute-intensive application, and therefore, a good offloading candidate.

In our experiments, we denote our preemptable code offloading approach as `PreOff`. We compare our approach to two basic non-preemptable code offloading approaches as implemented by related work (cf. Section V). Both approaches differ in the way they handle disconnections. The first basic approach (BasicOff-ReExec) starts a local re-execution of the offloaded method on the mobile device as soon as the connection breaks. Since no other safe-points than the initial safe-point at the time of offloading...
the method are available, the mobile device must re-execute the complete method locally again. The second basic approach (BasicOff-Wait) waits for a re-connection to receive the complete result of offloading. Besides these two basic offloading strategies, we also compared our approach to a fully-local execution not using any offloading (AllLocal).

B. Offloading without Link Failures

We start by comparing the different approaches in a scenario without link failures. In such a scenario, we expect PreOff to be less efficient than the other offloading approaches not using safe-points, because safe-pointing induce a certain overhead that only pays off when link failures occur. Therefore, a scenario without link failures gives us the possibility to evaluate the overhead of PreOff compared to BasicOff. Note that in a scenario without link failures, BasicOff-ReExec and BasicOff-Wait behave identically since their only difference is their reaction to failures. Ideally, this overhead should not affect the overall performance significantly.

Figure 3 depicts the power consumption of BasicOff over time for the Netbook and as a reference an all-local execution (AllLocal). Figure 4 depicts the Netbook results using PreOff. As can be seen from these two figures, BasicOff and PreOff have virtually the same power consumption in this scenario. Over the depicted period, PreOff spends 181.45 J vs 178.00 J for BasicOff, i.e., the overhead of PreOff is smaller than 2% compared to BasicOff. On the laptop, the absolute power consumption is higher for both approaches (PreOff: 253.85 J; BasicOff: 244.89 J) due to the smaller energy-efficiency of the Laptop (cf. Figure 5). However, the relative difference is similarly small (3.66%).

Figure 3 and Figure 5 also show for both devices the general benefits of offloading compared to an all-local execution. On the Netbook, the offloading starts at 14.59 s and lasts until 18.69 s, so a relatively long period is executed remotely (cf. Figure 3). As Figure 5 shows for the Laptop, the remote execution starts at 2.89 s and takes 4.43 s. Overall, AllLocal consumes 39.47% more energy in this scenario than BasicOff and 38.30% more energy than PreOff for the Netbook (40.12% and 37.93% for the Laptop).

So we can clearly state that even in a worst-case scenario with no link failures, the overhead of safe-pointing is very small. The reason for this is primarily the small number and size of received safe-points (cf. Algorithm 1). As a result, PreOff still saves a lot of energy compared to an all-local execution (AllLocal).

C. Offloading with Link Failures

Next, we consider scenarios with network link failures. First, we discuss the experiments with the Netbook device and six different scenarios. In this context, disconnections happen at different points in time, namely at $t_c = \{16.14, 16.31, 17.31, 18.54, 17.97, 19.02\}$. The time constraint $T_{\text{max}}$—until which a result should be returned—is set to 40 s, which is related to the time an all-local execution without offloading requires. After a disconnection, the network link remains unavailable for longer than the specified time constraint. Figure 6 depicts the results for the scenarios with disconnections at 16.14, 16.31 s, and 17.31 s, whereas Figure 7 shows the scenarios for disconnections.
Figure 6. Netbook: Power consumption of PreOff for three different scenarios with disconnections at 16.14 s, 16.31 s, and 17.31 s.

Figure 7. Netbook: Power consumption of PreOff for three different scenarios with disconnections at 18.54 s, 17.97 s, and 19.02 s.

For the laptop experiment, disconnections occur at 5.42 s, 6.15 s, 6.75 s, and 7.32 s (cf. Figure 8). Again, we assume that the disconnection period lasts longer than the required deadline, which is set to $T_{\text{max}} = 12$ s for these experiments. The results for the laptop experiments follow the same trend as for the Netbook. For short offloading periods until the disconnection, PreOff is more energy-efficient than BasicOff-ReExec. Indeed, only one intermediate safe-point leads to significantly smaller execution times (cf. $t_e = 5.42$ s with 11.46 s vs >12 s). Again, the energy and time savings are more pronounced for longer offloading periods and link failures that occur before the remote execution is completely finished. For instance, for $t_e = 6.75$, PreOff consumes 318.85 J with an execution time of 8.143 s. The basic offloading approaches are only more energy-efficient, if the link failure occurs after the offloading period, when the final result has already been returned to the mobile device (cf. $t_e > 7.32$ s in Figure 8). However, also in this scenario, where de-facto no link error occurred during the relevant offloading period, PreOff only consumes 3.5% more energy than BasicOff due to the overhead of receiving safe-points.

So overall we can state that safe-pointing significantly increases energy consumption and reduces execution time under link failures during offloading.

V. RELATED WORK

Code offloading approaches to improve the energy efficiency of mobile applications have been proposed in different related work. Cuervo et al. [4] proposed a fine-grained code offloading system called MAUI, which offloads methods to a remote server. MAUI uses integer linear programming to optimize offloading, also considering network parameters like bandwidth and latency. CloneCloud [1] utilizes so-called device clones running in the cloud for each device to automatically offload threads. Offloading is supported by runtime application profiling and static program analysis to define the partitioning of applications. A further prominent offloading approach from Giurgiu et al. [5]...
utilizes R-OSGi [6] to partition applications into modules, which can then be distributed between mobile device and remote server. Using a device and application profiler, an optimizer decides dynamically of migrate components between mobile device and server. Gitzenis et al. [7] also consider the local power management of mobile devices, e.g., by adjusting the processor speed, and focus on the trade-off between execution time and energy saving. The related decision problem is modelled with the help of a controlled Markov chain. The solution of the optimization problem is a policy, determining the execution site as well as the hardware power consumption. Although these approaches show the high potential of code offloading in terms of energy savings, they do not consider link failures and/or their adverse effects on energy and availability.

A first basic mechanism to also deal with link failures was introduced by Kwon et al. [8]. In their approach, the application programmer has to annotate methods that should be offloaded. Thereupon, the approach creates a single safe-point before offloading. Nevertheless, if a link failure occurs, the complete method is re-executed locally.

A second offloading approach that also considers link failures is the ENDA system proposed by Li et al. [9]. In this approach, methods can be offloaded to either small servers with WiFi access points called cloulets, or cloud servers via 3G mobile communication. Based on historic user traces, WiFi access points along the predicted route are chosen to connect the mobile device to cloulets. If no cloulets are available, 3G mobile communication is used to offload to a cloud server. If a link failure occurs during offloading, the mobile device simply waits for a re-connection.

Robustness was a focus of the work of Gordon et al. [10] who proposed the COMET offloading system, handling network and server failures. To this end, COMET utilizes Distributed Shared Memory (DSM) and Virtual Machine Synchronization techniques to synchronize offloaded threads. However, this fine-grained heap and stack synchronization induces high communication overhead. Moreover, COMET has to synchronize the complete initial state at the start of a mobile application (typically 750–810 kB). In contrast, instead of doing fine-grained synchronization, we only create intermediate safe-points at longer time intervals leading to much much smaller data to be transferred between the mobile device and the cloud environment.

VI. SUMMARY AND FUTURE WORK

In this paper, we proposed a novel code offloading approach to increase the energy-efficiency and responsiveness of mobile applications also in environments with network disconnectivity. To this end, we proposed the concept of preemptable code offloading, which allows for interrupting offloading after a link failure without losing the complete result calculated remotely until the failure and avoiding local re-execution of the complete offloaded code. Preemptable code offloading is based on safe-points containing all required information to continue local execution after a safe-point. The remote server creates safe-points and transmits them to the mobile device whenever it is beneficial in terms of energy. Although the transmission of safe-points induces communication and energy overhead, our evaluations show that this overhead quickly pays off in scenarios with link failures leading to significantly lower overall energy consumption on the mobile device. Moreover, safe-points also improve the application responsiveness by considering a maximum tolerable execution time under link failures.

As part of future work, we are going to further improve safe-point scheduling using, for instance, movement predictions to better predict impending link failures and minimize the number of required safe-points.

REFERENCES


