Resource and Execution Control for Mobile Offloadee Devices

Torsten Zimmermann, Hanno Wirtz, Jan Henrik Ziegeldorf, Christian Steinhaus, Klaus Wehrle
Chair of Communication and Distributed Systems, RWTH Aachen University
{zimmermann, wirtz, ziegeldorf, steinhaus, wehrle}@comsys.rwth-aachen.de

***Abstract*—Mobile offloading overcomes the resource limitations of offloader devices by splitting resource-intensive tasks and allocating subtasks to nearby offloadee devices. In processing its subtask, each offloadee effectively executes foreign and untrusted code which might both harm the device and exhaust its resources. Given the personal nature and constrained resources of offloadee devices, such as smartphones, precise control at the offloadee over the execution environment of offloaded tasks as well as the provided and consumed resources then is a natural requirement for the success of offloading approaches.**

We thus contribute a mechanism for fine-grained resource control of local task execution, benefitting allocation approaches by precisely assessing, advertising, and guaranteeing offloadee processing resources. Our design protects local device integrity and usability by isolating the execution of each task in a dedicated Linux container with precisely defined resource constraints. We highlight the performance and immediate applicability of our design through a prototypical implementation using LXC containers on COTS Android smartphones that achieves controllable task execution at minimal costs: Each container starts up in only 2 ms, imposes less than 5 % computation overhead, and consumes only 10 MB of memory.

I. Introduction

Mobile offloading harnesses the resources of nearby *offloadee* devices to overcome the resource limitations of *offloader* devices in the face of complex mobile applications, such as games, speech and image processing, and mobile computer vision [1], [2]. Existing offloading approaches [3]–[6] propose offloader-side mechanisms to manage and split computation tasks, discover nearby devices and their resources, as well as to allocate tasks to suitable devices. As a result, an offloadee that participates in mobile offloading is required to offer resources to as well as process said tasks, i.e., execute untrusted code from foreign devices. A natural requirement for the participation as an offloadee is then the ability to control this execution and to protect both the device and its resources.

However, offloadee devices are currently unable a) to assess whether the execution of a task may cause harm to the device and its data, and b) to precisely define and subsequently control the resources offered to and consumed by the execution of the task. This is because existing approaches [3], [5], [6] entirely abstract from execution at the offloadee, i.e., do not propose (or assume) any execution mechanism that would cater to offloadee device protection. Furthermore, existing approaches adopt a simplistic view of the resource availability at offloadees by one-time benchmarking of the general device capabilities. In this, they do not account the dynamics of local resource usage and availability as well as the actual preferences of the device owner with regard to the resources she wants to offer to offloaded tasks. We argue that, without a mechanism that accounts for the value and scarcity of offloadee devices as well as their owners, adoption of offloading approaches will fail, negating its proven benefits.

In this paper, we thus propose Resource and Execution Control for Mobile Offloadee Devices (RECMOD), a counterpart mechanism to existing offloader-side scheduling that realizes the requirements of offloadee device owners for participation in offloading approaches. Our prototypical implementation for Android devices builds on Linux Containers (LXC) and a set of adapted management tools. Thereby, RECMOD provides a feature-rich yet lightweight execution environment for offloaded tasks. In contrast to current approaches our design benefits offloading approaches by the following three main features:

Isolation of Execution: RECMOD executes each task within a dedicated container that strictly isolates it from the offloadee operating system and other offloaded tasks. Untrusted code is thus separated from the personal data of the device owner and is limited to processing without unforeseen side effects.

Fine-Grained Resource Control: We enable fine-grained resource allocation and control for each container, affording complete control over both the number of as well as the resources dedicated to (parallel) task execution. RECMOD further accounts for the dynamic availability of resources at mobile devices, e.g., in battery saving modes, and adapts both the advertising of resources towards the offloading framework as well as the allocation of local resources to containers. Our design thus supports the precise definition and subsequent guarantee of resource availability to the offloading framework, thereby enabling successful task allocation and performance of the overall offloading design.

Accounting for User Requirements: We introduce a practical offloadee-side approach that accounts for user requirements and control with regard to the execution of untrusted code and the instrumentation of valuable resources of a personal device.

Our design directly embeds itself into and benefits current mobile offloading approaches (Section II) by enabling the precise assessment of and the control over the computation, memory, and communication resources allocated to task execution (Section III). Crucially, using RECMOD, device owners are able to account for the respective capabilities of their devices and, in turn, control their contribution to resource provision in offloading approaches. We implement the container-based execution environment of RECMOD by adapting and extending LXC for Android devices (Section IV) and evaluate the performance, flexibility, and usability of our design along multiple proposed offloading applications (Section V). RECMOD provides a comprehensive offloadee solution that accommodates offloader, offloadee, and user requirements and at the same time contributes to the functioning of the respective offloading approaches (Section VI).

II. RELATED WORK

RECMOD provides a comprehensive mechanism for fine-grained resource control and isolation of computation in mobile offloading scenarios. In the following, we discuss existing approaches for mobile computation offloading and how RECMOD complements those. We then present a representative set of approaches that offload tasks to the cloud and discuss the differences and challenges between cloud and mobile computation offloading. Finally, we discuss existing approaches for isolation of task execution that, similar to RECMOD, utilize dedicated execution environments.

Mobile Computation Offloading: Offloading tasks among mobile devices strives to utilize nearby resources while reducing dependencies on infrastructure cloud environments and Internet connectivity [7]. Building on local, opportunistic contacts between resource-constrained devices, the challenges of mobile offloading revolve around the complexity management of partial tasks as well as their scheduling and distribution to adequate nearby devices. Representative for a multitude of approaches [3]–[6], the *Serendipity* framework [3] addresses these challenges by proposing different distribution algorithms as well as by allocating and disseminating tasks with a defined complexity only to devices that offer matching computational resources. Although authors of existing approaches unequivocally point out the need for user based control over the offered resources [5], they leave it to future work to provide actual guarantees for the announced resources.

RECMOD complements related mobile offloading frameworks by contributing the device-local part of mobile offloading, i.e., a mechanism for fine-grained resource control and task isolation. Specifically, RECMOD enables device owners to precisely control and report to other offloading frameworks the *guaranteed* resources offered to offloaded tasks, enabling an informed allocation of tasks. At the same time, RECMOD isolates computations and thus greatly reduces risks for owners of offloadee devices.

Cloud-based Computation Offloading: Originating from the work on *Cyber Foraging* [8], offloading of computationally expensive tasks to more powerful computation surrogates or the cloud promises the augmentation of constrained mobile device resources. Thereby, it enables the execution of complex, time-critical, and resource-demanding tasks [9]–[12]. Independent of the actual realization, cloud-based offloading revolves around the identification of suitable code parts that amortize the communication overhead required to offload to the cloud by the time and computation overhead saved. In this context, these works focus on the seamless migration of applications to the cloud [9], [11], as well as optimizing the elasticity and scalability of computation offloading [12].

Compared to mobile offloading, cloud-based offloading greatly differs in the individual communication models, i.e., external vs. local device-to-device, as well as in the availability and scalability of resources. Whereas cloud infrastructures scale easily, resources in mobile computation offloading are clearly limited and constrained by the number of available devices and their capabilities. RECMOD accounts for the challenges and characteristics of mobile scenarios and, in contrast to cloud-based solutions, especially for the inherent resource constraints of offloadee devices.

Dedicated Execution Environments: Regardless whether computation is offloaded to the cloud or to mobile devices, the type of execution plays an important role with regard to control and isolation of the task. In the aforementioned cloud computing scenarios, this is typically achieved via system-virtualization, utilizing Virtual Machines (VM). They either mimic the source system, i.e., Android [11], [12], or use code-portability across platforms, e.g., .NET Common Language Runtime [9], to execute code. A different approach is to provide the input for a computationally intensive task instead of the code, e.g., transfer an image or recorded sound file for further processing. Similar to cloud-computing, approaches for mobile computation offloading [13] also utilize code portability. Approaches like [3], [5], [6] built upon applications for the respective platform, e.g., Android, but without resource control. In this case, the underlying concept of process-level VMs, e.g., Dalvik or its replacement ART, already provides isolation, as each application is executed on behalf of a separate user inside a single instance of these VMs. As an alternative, the use of virtualization techniques such as processor emulation [14] or hypervisors for mobile platforms [15], [16] have been introduced. The latter use system-level virtualization to isolate tasks from each other, at the cost of running a full guest OS for each task. Lightweight virtualization architectures such as [17] built upon Linux namespaces to provide isolation between multiple *Virtual Phone* instances on a single mobile device, e.g., to distinguish between a business and private phone.

From our discussion of related work, we conclude that current approaches lack the practical support for user-defined resource control and isolation of tasks which limits their adoptability in real-life scenarios. In the following section, we derive certain challenges and requirements based on our analysis of related work, present our design and the core components, and how these complement current approaches.

III. DESIGN

RECMOD offers flexible, isolated, and resource-controlled execution of tasks on offloadee devices that participate in mobile-to-mobile computation offloading. Thus, it provides the missing piece to proposed offloading frameworks [3], [5], [6] that focus on offloader-side discovery, scheduling, and allocation mechanisms. Figure 1 illustrates the high level design of RECMOD and how it complements and augments existing

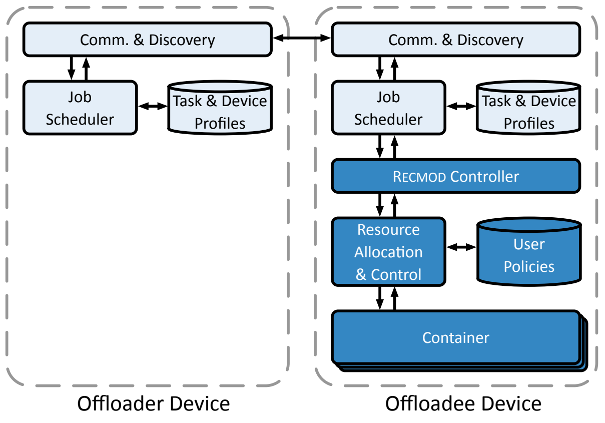

Fig. 1. Augmenting and interfacing mobile offloading frameworks, e.g., [3], [5], [6] (light colored), with RECMOD (dark colored). To offload a task, an offloader decides to request computation resources on nearby devices (Job Scheduler & Discovery) based on assessed task profiles (Task & Device Profiles). A potential offloadee replies with a resource offer based on configuration (User Policies) and current capabilities (Resource Allocation & Device Profiles). Subsequently, the task is transferred to the offloadee (Job Scheduler & Comm.) and is executed in an isolated execution environment, i.e., a container, configured and controlled as announced (Resource Control).

frameworks. We shortly outline how RECMOD interfaces with mobile offloading frameworks and detail each component in the respective sections.

Consider the following scenario: a mobile device, i.e., the *offloader*, initiates a resource intensive task. The *Job Scheduler* requests information about previous executions of the requested task and current computation capabilities (*Task & Device Profiles*). Having ascertained that local resources are insufficient to carry out the given task, the Job Scheduler decides to request resources at devices in the vicinity (*Comm. & Discovery*). On a mobile device that intercepts the request, i.e., the *offloadee*, the *Job Reception* i) checks if it is eligible and ii) what amount of resources can be offered for task execution (*User Policies*). If the current settings allow to accept a task, it replies to the request by announcing the resources. In case the announced resources match the task requirements, the offloader issues subparts of the task to the offloadee. Subsequently, RECMOD allocates the resources on the offloadee (*Resource Allocation*) starts the execution in an isolated execution environment, and monitors and controls the resource usage (*Resource Control*).

Based on the aforementioned exemplary scenario, participating as an *offloadee* thus raises four distinct challenges for a mobile device, which have been neglected in current approaches. *First*, in order to enable successful task allocation, an offloadee must precisely advertise and subsequently guarantee available resources towards the offloader, in spite of the naturally fluctuating resource utilization incurred by normal device usage. To this end, we enable the offloadee to set and control the limits for the resources, i.e., computation, memory, and communication, she wants to offer (Section III-A). *Second*, offering resources for offloaded tasks, as well as their subsequent execution, must not impede normal device usage and must thus be controllable by the device owner. Therefore, we allow the user to define descriptive policies (Section III-B). *Third*, execution of an essentially foreign, untrusted, and unchecked piece of code naturally requires protection of the personal device. We achieve this through the isolation of task execution in a dedicated container execution environment (Section III-C). *Last*, the active state of an offloadee device might change with regard to the resources made available, e.g., in battery-saving states or when restricting communication according to costs or data plans. In order to enforce the user-defined policies in these cases, our design adapts resource control to these varying device states (Section III-D).

We cumulatively address the aforementioned challenges by encapsulating each task in a LXC container instantiated on the offloadee. We thereby improve on existing mechanisms for offloadee-side execution in two aspects. First, current offloading frameworks [3], [5], [6] envision offloadee-side execution in running Android apps and thus do not afford direct control over the resources allocated to and consumed by each task. Second, while virtualization techniques for mobile devices [15], [16] offer full control and isolation, instantiating an entire OS per task introduces prohibitive storage, memory, and computation overhead [17]. In contrast, our design strives for a lightweight solution that combines the performance of native execution with the isolation and control functionality afforded by virtualization approaches, motivating our selection of Linux containers as the task execution environment in RECMOD. In the following, we illustrate the details as well as the benefits of this design.

A. Resource Allocation and Control

At its core, RECMOD enables users to precisely control the resources offered to the execution of offloaded tasks. To this end, it provides, for each accepted task, a dedicated container and allocates a defined set of resources to it and thus the execution of a task within it. In the following, we thus refer to resources allocated to a task and to its container interchangeably.

Intuitively, the resources offered by a device and consumed by a task are computation, memory, and communication. We make no assumption about the prevailing resources that might be requested in offloading scenarios due to the diverse nature of tasks that benefit from offloading. For example, speech and face recognition are computation-bound tasks and thus predominantly require CPU resources at offloadees [3]. In contrast, tasks that convert speech to text, or vice versa, require substantial communication resources as well as dynamic memory on the offloadee device. We thus treat each resource, as well as possible combinations, equally and separately. RECMOD then enables the instantiation of execution environments with a given set of resources, as well as the dynamic adaptation of those to current device states based on resource control. We summarize the resources that can be allocated and controlled by RECMOD in Table I. In the following, we detail the challenges of providing each resource in mobile to mobile computation offloading and how we tackle this within RECMOD.

Computation: Modern mobile devices are equipped with multi-core CPUs, e.g., quad- or even octa-core. Additionally, mobile devices typically are able to control and regulate their CPU

Resource	Control offered by RECMOD
Computation	· accessible # CPU cores · fixed lower and upper limit of CPU share
Memory	· upper limit
Communication	· type of interface (Wi-Fi, Cellular) · traffic limit for up- & download · bandwidth throttling for up- & download

TABLE I
OVERVIEW OF RESOURCES THAT CAN BE ALLOCATED BY RECMOD, AS WELL AS RESPECTIVE CONTROL MECHANISMS.

through frequency scaling or even powering off single cores to save battery when computation power is not needed or if the battery level falls below a certain threshold. Exemplary, a Nexus 5 running stock Android 5.1.1 powers off three of the four available cores and sets the frequency to the lowest level when the screen is locked and no high priority task is running. Depending on the manufacturer, these settings are either controlled by the OS kernel itself or a system service.

However, to gain full control over the assignment of computation resources, we enable RECMOD to override and control CPU settings based on user policies, which we discuss in Section III-B. RECMOD is enabled to assign a single or multiple of the available cores to the offloaded task. Additionally, we also allow to set lower and upper limits for the CPU usage, i.e., the time share on each available core. In this, the lower limit guarantees a minimum of service to the offloader device and the upper limit protects the offloadee device from unbounded or monopolizing resource consumption by the task. Once a task has been accepted for execution, we do not permit to decrease the lower limit or to scale down the CPU frequency as this would violate the guarantees made to the offloader.

Memory: The memory assigned to the offloaded task is a second crucial resource, due to the *transient* and *result-oriented* nature of tasks offloaded between mobile devices which favors in-memory data and processing over persistent long-term storage. In RECMOD, we decide to set the available memory for the task to a fixed level that may be increased based on user preferences but never be decreased. This allows providing *worst-case* guarantees that enable more precise job scheduling decisions by the offloader.

Controlling the memory in a dynamic fashion, i.e., allowing it to grow and shrink adaptively, is infeasible with regard to our scenario for various reasons. First, when memory has been decreased for a running task, we cannot assure to have enough memory available to just increase on demand, as the user or applications might interact with the offloadee device. We stress that the top priority is to not disturb and harm the normal usage of the mobile device. Second, even if there is no interaction or concurring application, we envision to allow the offloadee device to accept multiple tasks in parallel. Therefore, a fixed and guaranteed upper limit thus enables the management of multiple tasks and allows to keep a certain amount of memory free to accommodate user applications and tasks not directly controlled by RECMOD.

Communication: Some tasks might require communication for *local* exchange of commands, intermediate results with the offloader, or to retrieve *external* service data located in the Internet. Making the respective communication interfaces of the offloadee, i.e., Wi-Fi or 4G, available to the task execution incurs different types of costs for the offloadee. First, all of the aforementioned communication interfaces consume different amounts of energy. Moreover, offloaded tasks may interfere with other applications' traffic and thus harm the overall user experience. Lastly, allowing use of the cellular interface will introduce either monetary costs or limit the data plan traffic left for the user.

While we are able to guarantee exact computation resources and available memory, guaranteeing certain amounts of bandwidth is infeasible since we are not in control of the communication infrastructure. Instead, depending on the setting (cf. Sec. III-B), we allow environments to access these communication interfaces, to specify a traffic limit in both directions, and to throttle bandwidth in the upload direction. In addition, we allow to set what kind of interface and what kind of traffic, e.g., local or external, is allowed.

In the following, we illustrate how we allow device users to control the resources and which settings are feasible in the scenario of mobile computation offloading.

B. User Preferences & Task Control

RECMOD enables users to share their available device resources in mobile offloading scenarios in a *controlled* manner. As the basis for such sharing, we envision users to a priori define the *overall* resources they want to make available to offloaded tasks, in view of their normal device usage and their sharing preferences. The resulting *resource pool* (cf. Tab. I) is then made available to the embedding offloading framework by RECMOD, which then can be assigned to individual tasks as requested by an offloader. Moreover, user configurations can be bound to constraints such as a certain battery level, thus disabling to advertise the availability of multiple cores or not allowing communication. For communication traffic limitation, the allowed traffic can be coupled to the amount of traffic left in the user's monthly data plan, if we consider a cellular connection. As these settings can be applied dynamically and constraints only need to be evaluated when a nearby offloading device requests resources, this only adds marginal management overhead to the overall offloading framework.

An important aspect is how RECMOD handles situations where tasks are reaching the limits of the provided resources. Ultimately, this decision is up to the embedding offloading framework and the implemented *Job Scheduler/Reception* (cf. Fig. 1), as the task execution planning is not part of RECMOD. In the following, we sketch how RECMOD supports the control of tasks. Independent of the resource, the Resource Control component can signal the task and inform about the approaching limit, such that direct countermeasures could be taken by the task, depending on the resource and implementation of the task. If the task is reaching the provided limit for communication, i.e., amount of traffic in either direction, computation is still possible. However, if the task is computing at its resource limits for a certain amount of time, we propose the following solution. As RECMOD is monitoring the usage of the provided resources, it will recognize the aforementioned behavior. When the current device state and user policy permits, RECMOD can increase the already provided resources. This solution does not require any changes to the task itself, e.g., including an API to request more resources. When no more resources are available or an increase is not allowed, it depends on the implemented Job Scheduler, if the execution should be paused or stopped.

C. Isolation of Tasks

In mobile offloading scenarios, users share their resources with others and accept to execute foreign code on their devices. This bears the risk of executing code that is malicious or harmful to the offloadee device, e.g., via non-authorized access to private user data. Without adequate countermeasures, this prevents any user acceptance and the real world applicability of mobile offloading schemes. Thus, RECMOD isolates the task in a dedicated container (cf. Fig. 1) to delimit its execution effects and protect offloadees. In the following, we present what is protected by use of containers in RECMOD.

First and foremost, erroneous code or inputs that may cause a crash of the application must not cause a crash of the device operating system. Moreover, an offloaded task should not be able to access resources that have not specifically been provided for the task itself. Besides the resources RECMOD allocates and assigns to the container itself, such resources also include data or memory of other applications. In RECMOD, the isolated task has no notion about tasks and processes outside of its container and thus is only able to control and monitor processes triggered by the task for its execution. This also includes access to personal data or devices, e.g., phone contacts, pictures, messages or the usage of the camera or microphone.

From the perspective of the offloadee device, the execution of the task itself in the container is fully controllable in terms of starting and stopping their execution as well as the allocation of resources. To accommodate tasks in a flexible way, we do not limit the execution of tasks in RECMOD to a specific type of implementation, e.g., self-contained binaries or scripts. Thus, we offer basic functionality for a variety of task implementations that can be extended according to future needs. We refer to Section IV for the actual implementation and management of containers in RECMOD.

D. Accounting for Varying Resources

Once an offloader has requested resources for computation, potential offloadees in the vicinity may advertise their current computation capabilities. In this regard, RECMOD utilizes its possibility to control resources and thereby the ability to advertise a *guaranteed* set of resources, either based on specific profiles or on user policies (cf. Sec. III-B). However, these capabilities are dynamic and may vary. In the following, we illustrate how to handle variations in a RECMOD-enabled offloading framework using multiple device profiles.

As briefly discussed in Section III-A, computational resources on mobile devices do not stay stable per se. For example, a certain battery level or device state might cause the mobile device to lower the CPU frequency or to even shut down cores. Moreover, applications running in parallel, user interaction, and multiple parallel containers also impact the availability of resources for offloaded tasks. In addition, user-defined policies (cf. Sec. III-B) add further possibilities, such as to not offer communication resources below a certain battery level or to limit traffic via the cellular interface.

Because similar user-defined policies are also imaginable for CPU settings or memory, the overall amount of possible resource settings is increased even further. Therefore, advertisements of resources need to be updated on a regular basis. To cope with this variety of device states the profiling step of previous approaches [3], [9], [11] is extended. We propose to profile tasks with respect to available device states and user-defined policies. To this end, device profiling includes executing pre-defined benchmarks or exemplary tasks with varying input on a set of *individually* configured execution environments as part of a boot strapping process. In a real world setting, generating new device profiles as soon as new user-defined preferences are provided is not feasible, due to the aforementioned possibly huge amount of policies. As a practical solution, when announcing resources during discovery, the offloadee should announce the available resources according to the user policy but include the best matching device profile based on the previously described process.

The resulting profiles in combination with the ability to allocate resources (cf. Sec. III-A) allow existing schedulers (e.g., [3]) to even distribute tasks with certain level of time constraints, as RECMOD guarantees the availability of the announced resources.

IV. IMPLEMENTATION

In the following, we describe and discuss our prototypical implementation of RECMOD. In recent years, operating-system level virtualization based on containers such as LXC[1] or Docker[2] emerged and have established themselves as valuable alternatives to full system virtualization approaches. The concept of containers is enabled by features of the Kernel, i.e., *cgroups* and *namespaces*. The former allows the limitation, accounting, and isolation of resources for a set of processes, and the latter provides an abstraction of global system resources, e.g., providing a fresh set of unique process IDs per namespace. This kind of virtualization allows to realize an application or system container, both using the *underlying* host kernel, which renders them leaner and smaller than the aforementioned hypervisor-based virtualization approach, and showing performance similar to native execution of the task [18]–[20].

We base our prototypical implementation of RECMOD on LXC. Our prototype is realized on an LG Nexus 5 (*hammerhead*) smartphone running a rooted Android 5.1.1,

[1]https://linuxcontainers.org/
[2]https://www.docker.com/

with a Kernel 3.4. Inside the container, we use Debian 8 (*jessie*), that uses the underlying host Kernel, to provide a rich and extensible execution environment, e.g., including well established libraries and script interpreters. Container structures in current Kernel versions already support a full-fledged execution environment that offers isolation of processes and basic resource control, rendering them a perfect match for the requirements of offloadee task execution as outlined in the previous sections. Moreover, more recent versions promise additional features that further enhance the capabilities of encapsulating tasks within container structures, such as mapping the root user in the container to a less privileged user outside the container. In terms of protection, e.g., compared to system-virtualization techniques such as [15], [16], there is a trade-off in using LXC containers. Because the Kernel is shared between the container and the host, it does not protect against exploits of kernel bugs. While system-virtualization techniques solve this, though instead exploitation of hypervisor bugs might be possible, the requirements for a lightweight approach led us to choosing LXC over VMs. This can also be mitigated with support offered by more recent Kernel versions.

Next, we outline how the resources offered by an offloadee are controlled in our prototypical setup. In the container itself, we create a computing user with limited permissions, that we map to a user added to the Android host system. A major benefit is that we can monitor and control the traffic using *iptables* and *tc* based on the respective User ID (UID) outside the container. For monitoring of other resources, we add a set of scripts that control the LXC tools according to provided resource control and user policies. Management for the CPU based resources, i.e., computation time and number of cores, is completely manageable via LXC, i.e., the assigned resources are guaranteed. In case of memory, slightly more adaptations are necessary. Although LXC allows to set an upper limit, this is not guaranteed to be available as memory cannot be blocked for the container. To this end, we allow RECMOD to override the internal Android memory management settings, i.e., the Low Memory Killer (LMK). The task of the LMK is to kill processes based on defined priority values and minimum free memory settings. Changing both (dynamically), allows us to keep some memory free based on user-based preferences serving as headroom for containers and to prevent the LMK from interfering with the container's memory.

V. EVALUATION

In this section, we first evaluate the overhead introduced by RECMOD's LXC based execution environment, using exemplary tasks such as face detection and speech-to-text, inspired by [3]. Second, we illustrate the resource control between containers themselves and between containers and user applications. Finally, we evaluate the scalability of the proposed approach using an exemplary task and multiple containers in parallel and discuss resource limitations.

As RECMOD provides resource control and a general execution environment for existing offloading and scheduling approaches (cf. Sec. III), we focus our evaluation on the aforementioned aspects on the offloadee device. If not stated otherwise, we conducted 30 independent runs of each evaluation and show the mean and the 99 % confidence interval.

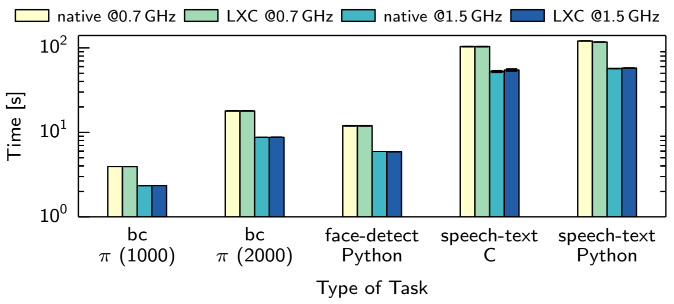

Fig. 2. Comparison of computation times of exemplary tasks being executed natively or in a chroot environment (in the case of Python implementations) and inside a container at different frequencies. Shown are the means and the (hardly discernible) 99 % confidence intervals.

A. Overhead

As discussed, an execution environment in the realm of mobile task offloading should add as little overhead as possible. We evaluate the impact of RECMOD's container based solution on task completion as well as memory requirements and show that our prototypical implementation only adds marginal overhead to the overall task itself. Moreover, we analyze the delay for starting a task in a dedicated container and find that we can provide a configured container and subsequently start a task within a few milliseconds. During our overhead and scalability evaluation, we began by setting the CPU frequency to the highest available level, 2.2 GHz. However, due to excessive subsequent tests, the device overheated and the CPU frequency was automatically reduced to protect the hardware. Thus, we limit the CPU frequency to a maximum of 1.1 GHz if not stated otherwise, based on empirical test, in order to obtain stable results even in multiple successive evaluation runs. If not stated otherwise, the smartphone is not executing other applications than RECMOD and the task(s) in control.

CPU and Memory Overhead: To evaluate the impact of the container based approach of RECMOD on the computation, we implement several tasks and compare overheads for running them *natively* on the device and in the *container*. The goal of this evaluation is twofold. First, we want to analyze the impact of RECMOD to the mobile device itself, as the overall capability of the offloadee and the usefulness as an offloading target should not be diminished. Second, we want to gain an comprehensive understanding of actual resource requirements of the aforementioned tasks and their applicability on mobile devices. Inspired by the evaluation in [3], we implemented exemplary face detection and speech-to-text tasks. The former is a Python implementation based on *OpenCV*. For the latter task, we use two implementations, one in Python and a second in C which are both based on *PocketSphinx*. Moreover, we compiled the GNU *bc* tool and calculate different amounts of decimal places of π. To run the Python-based implementations outside a container, we `chroot` in the respective `rootfs`. We repeat these settings with the CPU speed set to 0.7 GHz and 1.5 GHz.

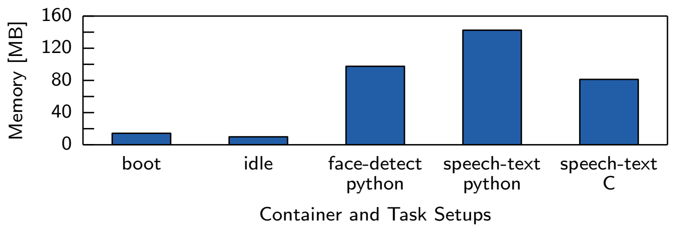

Fig. 3. Memory usage of a container running a Debian 8 in various setups. During boot, the memory usage is around 15 MB and 10 MB when idling. Running various tasks in the container increases memory to a range of 81.25 MB up to 142.45 MB.

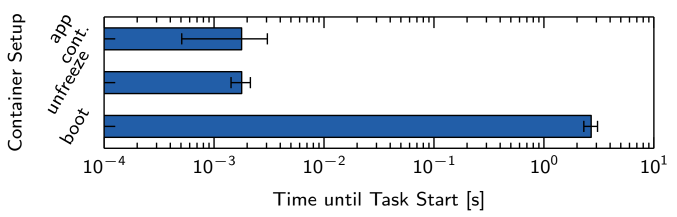

Fig. 4. Time until a task can be executed. Starting a container running a Debian 8 from scratch, i.e., booting and then starting the task, is denoted as *boot*. Starting a task in an already established but freezed container is denoted as *unfreeze*, and starting the task in an application-only container as *app-cont.*

Figure 2 shows the mean with 99 % confidence interval of the run times of the aforementioned tasks over 30 independent runs. Executing the *bc* tool natively is slightly faster (below 1 %) than in the container for both CPU frequency settings. Performing the face-detection task on a picture with a resolution of 2560 × 1536 finishes after 4.94 s when executed natively and after 4.91 s in the container at 1.5 GHz (10.95 s and 10.96 s for the lower CPU speed). Running the C-based speech-to-text task nativley at 1.5 GHz and 0.7 GHz on a 30 s `wav` file takes 51.37 s and 102.52 s, respectively. The Python-based implementation runs natively in 55.93 s and 119.52 s. In the container, the C-based task takes on average 53.67 s and 102.45 s, while the Python-based implementation runs in 56.64 s and 116.11 s. Summarizing, the experienced computation overhead, if any, is at most 5 % in all of the evaluated tasks.

To quantify the memory overhead, we monitor the memory usage of a container during boot, while idling, and while executing several exemplary tasks. The maximum memory usage is acquired by the Kernel's cgroup interface. Booting a container running Debian 8 on top of the host Kernel yields an maximum memory usage of 14.28 MB, which decreases to 9.89 MB after the boot process (cf. Fig. 3). Again, we stress that we do not decrease the assigned memory to a container once it is running. Thus, the minimal amount of memory that has to be allocated by RECMOD has to fit the requirements of a container during boot. For the exemplary tasks, the maximum amount of memory consumed is in the range of 81.25 MB up to 142.45 MB. Please note that the memory required by bc and a container is around 11.6 MB and therefore covered by the allocation needed for a container to start.

Time Overhead: Next, we evaluate the delay until a task can be started after is has been received by the offloadee, i.e., instantiating and starting a container, and finally executing the task after the boot process. If this time exceeds a certain limit, the benefit for the offloader might be limited or even lost, depending on the overall execution time it would take to run the task locally. On average, this takes 2.68 s on the Nexus 5 (cf. *boot* in Fig. 4). Depending on the task, referring back to Figure 2, this might add substantial overhead compared to the execution time, e.g., 55 % in the face-detection use case. To further reduce this time, we utilize techniques offered by LXC and instantiate and completely boot a container without further configuration and subsequently *freeze* it. Thus, it will only consume a marginal amount of memory (9.89 MB, cf. Fig. 3), but no processing is performed. Upon task reception by the offloadee, the container is configured regarding the announced resources, the task is handed over to the container, and finally the container is *unfreezed* to start the execution. Using this approach, the time to start a task is drastically reduced to 1.8 ms (cf. *unfreeze* in Fig. 4). Depending on the configuration and available resources on the device, i.e., the ability to execute more than one container at a time, the aforementioned step can be repeated and keep another paused container for an upcoming task, with marginal overhead. Finally, LXC allows to start an application in a container (*app-cont.*), *without* the need to boot and start an additional system on top of the Kernel (Debian 8 in our case), which shows a similar time as to unfreeze a container. However, in our prototypical implementation and the lack of certain Kernel features (cf. Sec. IV), we use the former approach for the complete evaluation.

Storage: In our prototypical implementation (cf. Sec. IV), we run Debian 8 inside the container to realize a flexible execution environment. The size of the base `rootfs` is 106 MB. In this base configuration, the container is able to execute simple bash scripts, i.e., using the bash-builtins. When adding functionality, e.g., a Python interpreter or the essential build tools to compile C/C++, this grows to 182 MB and 297 MB respectively. During our use case evaluation, i.e., speech-to-text and face-detection (Python based), the necessary storage increased to 431 MB and 554 MB respectively. To keep the storage overhead at a reasonable level, especially when allowing to run multiple containers in parallel, one could use the *same* `rootfs` for several container instances. Upon start, RECMOD then assigns a different computation user to each container, thus providing an individual `home` for each task. Note that this does not impact the resource allocation and control.

B. Resource Control & User Policies

A core feature of RECMOD is the fine-grained control over resources based on user policies. In the following, we illustrate how exemplary policies affect the computation between containers themselves and between user applications and containers.

First, we consider a user policy that allows the offloading framework to instantiate two containers for task execution. Both are assigned to the same CPU core. However, one container (C_1) is assigned at least 75 % of the CPU core and the other container (C_2) is allowed to use at least 25 %, as long as both

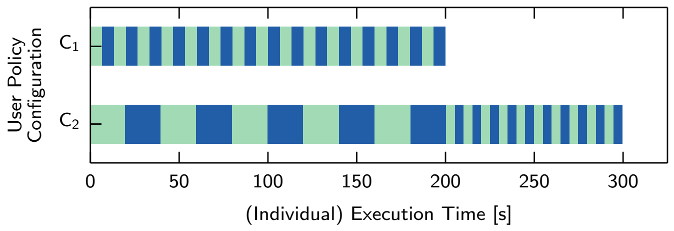

Fig. 5. *Two parallel* containers on *one* CPU core executing 30 runs (altering colors) of the face-detect task in an iterative manner. User policy for this setting is as follows: as long as *both* containers (C_1, C_2) are running, C_1 gets 75 % of the core as a guaranteed lower limit and the rest is assigned to C_2. As soon as C_1 is finished, C_2 is allowed to monopolize the complete core.

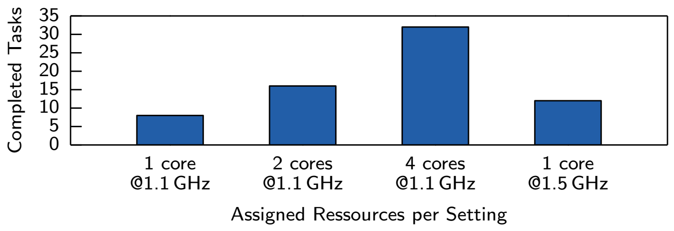

Fig. 6. Repetitive execution of *four* parallel instances of the face-detect implementation in a *single* container. After 60 s, we count the total number of finished tasks and vary the available computation resources, i.e., number of cores and frequency, and repeat the previous step.

are executing tasks. To achieve faster task completion when resources are available again, the user allows a *single* container to use as much CPU shares as available on the core. To illustrate the effect, we again use the face-detect implementation and execute successive runs on the same input image in C_1 and C_2, which are controlled by RECMOD respecting the aforementioned user policy. As depicted in Figure 5, in the presence of two containers, the execution of individual tasks in C_1 is faster than in C_2. On average, a task in C_1 requires 6.67 s to finish and 20.09 s (ratio of $\approx$ 3:1) in C_2. As soon as C_1 has stopped, C_2 is able to finish the execution with an average of 4.94 s for the remaining tasks as it is able to increase its resource allocation based on the aforementioned user policy.

Next, we allow varying the available computation resources, i.e., number of cores and frequency, assigned to a single container that is executing multiple tasks in parallel. We therefore assign a single CPU core, set to 1.1 GHz, to the container and start *four* instances of the same task, i.e., repetitive face-detection on the same image, overall resulting in 8 completed tasks in a time window of 60 s. Subsequently, we repeat the run as before but assign a second core (16 tasks) and four cores (32 tasks) respectively. Finally, we restrict the container back to one core but increase the CPU frequency to 1.5 GHz, allowing to complete 12 tasks in the given time (cf. Figure 6). In a real-world setting, this variable control of computation resources is feasible when the system has free resources again or limit resources when user space applications are started.

Policies that enforce the first example, i.e., increase on demand, are also possible for communication and memory, as discussed in Section III-B. In contrast, the second example

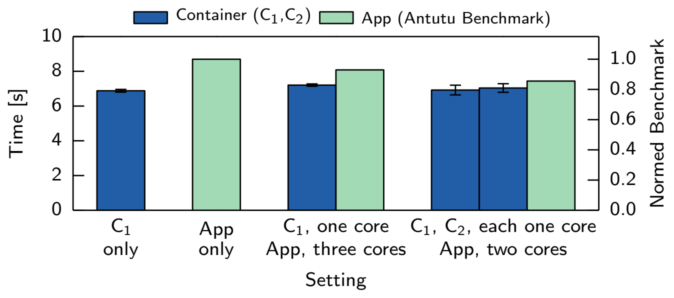

Fig. 7. Running a user space app, in parallel to containers, that run the face-detect task. First, both run exclusively. Next, the number of cores for user space apps is decreased and cores are assigned to containers. While benchmark scores decrease (right axis), the task completion time of the container stays stable. Variations are due to system processes on the respective core.

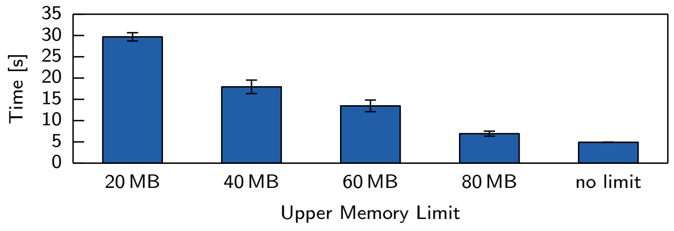

Fig. 8. Face-detect task completion time when swapping has to be performed due to memory limits. To this end, we enable swap in the Kernel and run a single container with different limitations.

only yields feasible policies for computational resources. This is because memory can not be revoked, i.e., if extra memory is assigned it might immediately be allocated by the task and thus can not be reduced without interfering with the tasks functioning. Therefore, we opt against the ability to reduce it again, as this would require additional signalling and management by the task itself.

In the following, we evaluate the effect of a running container on user space applications, depicted in Figure 7. The user space application is represented by *Antutu* Benchmark, that comprises, among raw CPU and memory benchmarks, 3D rendering representing games and user experience benchmarks. As a baseline, we let the Antutu Benchmark run on four cores set to 1.1 GHz, without anything else than the base system running in parallel. Subsequently, we assign only three and two cores to the user processes running on the smartphone. The remaining cores are assigned to individual containers running the face-detect implementation. We show the maximal achieved normed score of the benchmark, which decrease by 8 % for each core we remove from the resources assigned for user space applications. The containers running face-detect stay rather stable between 6.88 s and 7.21 s, due to the assigned resources. Slight variations are due to the load of system processes running on the respective cores.

C. Scalability

In our previous evaluation setups, we have showed that it is possible to execute multiple containers (cf. Fig. 5) on a COTS smartphone. Due to fine grained resource control offered by RECMOD, we allow an offloadee device to accept more than one task at a time to be executed in an individual container, if resources permit. In our prototypical implementation, this is mainly influenced by the overall available computation and memory resources and usage by the underlying operating system, i.e., Android, itself. To obtain a better understanding of the scalability, we assume an idling smartphone with no user interactions or applications interfering. Whereas the CPU is only experiencing minimal load and therefore possibly scaled down, slightly more than 50 % of the available 2 GB memory of the smartphone is occupied by the operating system (cf. Android Compatibility Definition Document [21]). Moreover, the stock Android 5.1.1 Kernel 3.4 is configured *without* swap support, making the remaining memory a hard limit. We were able to start six containers in parallel, configured to run on three cores, on an idling smartphone that run the face-detect task, only constrained by the systems memory. Please note that during our evaluations and depending on the density of the tasks, i.e., multiple iterations at a certain frequency for minutes without any pause, the smartphone began to heat up, which lead to thermal events that forced the CPU to scale down. Therefore we set the assigned CPU cores to a moderate level that allows to run excessive tests without heating problems, which is, based on empirical test, 1.1 GHz on the LG Nexus 5.

Of course, in a real world setup and depending on the respective user-policy affecting the memory management (cf. Sec. IV), accepting a relatively large number of memory intensive jobs blocks the device for normal usage. To overcome the memory limitation and to evaluate the effect of swapping to the task completion time, we activate swapping in the Kernel and provide a swap file of 1 GB. We again use the face-detect task and start a container with different memory limits, such that swapping has to be performed. The results are illustrated in Figure 8. In the case of a 20 MB memory limit, the completion time increases by a factor of ≈ 6 compared to the case of unlimited memory.

VI. Conclusion

In this paper, we have presented RECMOD, an approach that complements existing mobile computation offloading frameworks by controlling, and in turn being able to guarantee, the precise amounts of local resources available for offloaded task processing. Based on such guarantees, an offloading device is able to efficiently schedule and allocate tasks to offloadee devices. In turn, an offloadee device, is able to allocate controlled amounts of resources for offloaded tasks, following the resource policies defined by the device owner. Moreover, based on LXC, RECMOD provides an isolated execution environment for the offloaded task.

Our prototypical implementation and evaluation on a COTS Android smartphone based on Linux containers illustrates the applicability on current mobile devices, adding only marginal overhead in computation ($< 5\,\%$) to the execution of the offloaded task itself compared to native performance. Moreover, it allows for fine-grained resource control between multiple offloaded tasks themselves as well as between tasks and user space applications. Utilizing LXC, we are able to provide a freezed container for an upcoming task, with marginal memory requirements ($< 10\,\text{MB}$) while paused, that is ready for task execution within only 2 ms.

As possible extensions, we envision RECMOD to offer self-learned policies that incorporate assessed current and past device states to decide if computation should be offered, e.g., based on device states or user behavior [22]. Moreover, the resource control and isolation of RECMOD is not limited to the use in mobile device computation offloading, but can also be utilized for easier realization and faster adoption of crowd computing approaches such as [23], [24] or crowd-assisted mobile measurements [25].

Acknowledgements

This work has been funded by the DFG as part of the CRC 1053 MAKI.

References

[1] W. Hu *et al.*, “The case for offload shaping,” in *ACM HotMobile*, 2015.
[2] P. Jain, J. Manweiler, and R. Roy Choudhury, “Low Bandwidth Offload for Mobile AR,” in *ACM CoNEXT*, 2016.
[3] C. Shi *et al.*, “Serendipity: Enabling Remote Computing Among Intermittently Connected Mobile Devices,” in *ACM MobiHoc*, 2012.
[4] C. Shi *et al.*, “Computing in Cirrus Clouds: The Challenge of Intermittent Connectivity,” in *ACM MCC*, 2012.
[5] K. Habak *et al.*, “Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge,” in *IEEE CLOUD*, 2015.
[6] G. Calice *et al.*, “Mobile-to-mobile opportunistic task splitting and offloading,” in *IEEE WiMob*, 2015.
[7] D. G. Murray *et al.*, “The Case for Crowd Computing,” in *ACM SIGCOMM MobiHeld*, 2010.
[8] R. Balan *et al.*, “The case for cyber foraging,” in *ACM SIGOPS*, 2002.
[9] E. Cuervo *et al.*, “MAUI: Making Smartphones Last Longer with Code Offload,” in *ACM MobiSys*, 2010.
[10] M. Satyanarayanan *et al.*, “The Case for VM-Based Cloudlets in Mobile Computing,” *IEEE Pervasive Computing*, 2009.
[11] B.-G. Chun *et al.*, “CloneCloud: Elastic Execution Between Mobile Device and Cloud,” in *ACM EuroSys*, 2011.
[12] S. Kosta *et al.*, “ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *IEEE INFOCOM*, 2012.
[13] G. Huerta-Canepa and D. Lee, “A Virtual Cloud Computing Provider for Mobile Devices,” in *ACM MCS*, 2010.
[14] M. Black and W. Edgar, “Exploring mobile devices as Grid resources: Using an x86 virtual machine to run BOINC on an iPhone,” in *IEEE/ACM GRID*, 2009.
[15] K. Barr *et al.*, “The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?” *ACM SIGOPS Oper. Syst. Rev.*, 2010.
[16] C. Dall and J. Nieh, “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor,” in *ACM ASPLOS*, 2014.
[17] J. Andrus *et al.*, “Cells: a virtual mobile smartphone architecture,” in *ACM SOSP*, 2011.
[18] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *IEEE IC2E*, 2015.
[19] A. Machen *et al.*, “Migrating running applications across mobile edge clouds,” in *ACM MobiCom*, 2016.
[20] W. Felter *et al.*, “An updated performance comparison of virtual machines and Linux containers,” in *IEEE ISPASS*, 2015.
[21] Android Open Source Project, “Android Compatibility Definition Document,” [Online, accessed 12/12/2016] https://source.android.com/compatibility/cdd.html.
[22] S. L. Jones *et al.*, “Revisitation analysis of smartphone app use,” in *ACM UbiComp*, 2015.
[23] J. Cappos *et al.*, “Seattle: a platform for educational cloud computing,” in *ACM SIGCSE*, 2009.
[24] M. Y. Arslan *et al.*, “Computing while charging: Building a distributed computing infrastructure using smartphones,” in *ACM CoNEXT*, 2012.
[25] A. Nikravesh *et al.*, “Mobilyzer: An open platform for controllable mobile network measurements,” in *ACM MobiSys*, 2015.