technische universität
dortmund

Bachelorthesis

Saving Energy in Embedded Systems Using Nearby Resources

Alexander Starinow
07 August 17

Supervisors:
Prof. Dr. Jian-Jia Chen
Dr-Ing. Anas Toma

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Contents

Abstract

This thesis evaluates the energy consumption of a device that executes tasks locally and uses auxiliary resources.
Mobile embedded systems are widespread in today's society and are used for all kinds of purposes whatsoever. However, these embedded systems have limitations with respect to processing power and battery life, due to the requirement of being mobile. To overcome these limitations these systems communicate with servers to save energy and increase computation speed. In the setup in this thesis, the remote execution of a task on a server can theoretically reduce the consumption of energy by 99%.
But there are some challenges when communicating with a server. A network can be overloaded and therefore have a low bandwidth or being not reachable. This thesis proposes a new model that uses Nearby Resources as Auxiliary Resources. This model will be evaluated on real hardware using different tasks as examples for real-life applications. Using Nearby Resources can save up to 98% per task and also reduce the workload on remote servers and the relieve the bandwidth.

1 Introduction

Mobile embedded systems are very widespread today. In the year of 2015 there were 1.86 billion smartphone users worldwide and their number is expected to reach to reach 2.87 billion by 2020.[1] Furthermore, mobile embedded systems like smart watches, smart glasses, cameras, drones, virus scanners and infrared cameras are used worldwide. These systems are more and more capable of running a wide range of applications, which demand increasing computational power and consume a lot of energy. Yet because they are required to be mobile, they are limited in battery life and computational capabilities amongst other things. To overcome those limitations they can use mobile cloud computing, as a form of computation offloading, to offload parts of applications [8]. Computation offloading makes it possible for mobile embedded systems to reduce energy consumption, increase performance and even running applications while the system itself is unable to run at all [9].
Offloading in use today is already known to reduce energy consumption and improve performance in performing certain applications. Sometimes the offloaded data needs to be pre-processed for offlading to be favorable. Face recognition is only one example how mobile embedded systems benefit from computation offloading [12].

Contribution

In this thesis literature on computation offloading is reviewed. The experiment evaluates the use of a middleware server in a real environment. For the evaluation I made experiments using ODROID XU4s and wrote scripts to automate benchmarks, monitoring power and energy consumption.

1.1 Motivation

Computation offloading allows mobile embedded systems to run an even wider range of applications. These applications can be mathematical calculations, like a multiplication of large matrix tables, image or video processing, for example translating a text from a photo or transcoding video files, and artificial intelligence in games [8, 9, 10]. Running such applications on a resource-rich mobile cloud can save energy and time on the client

[1] Number of smartphone users worldwide from 2014 to 2020 (in billions), Statista, https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

system. As can be seen in figure 1.1 executing a task on a remote server can speed up the execution and possibly also save energy.

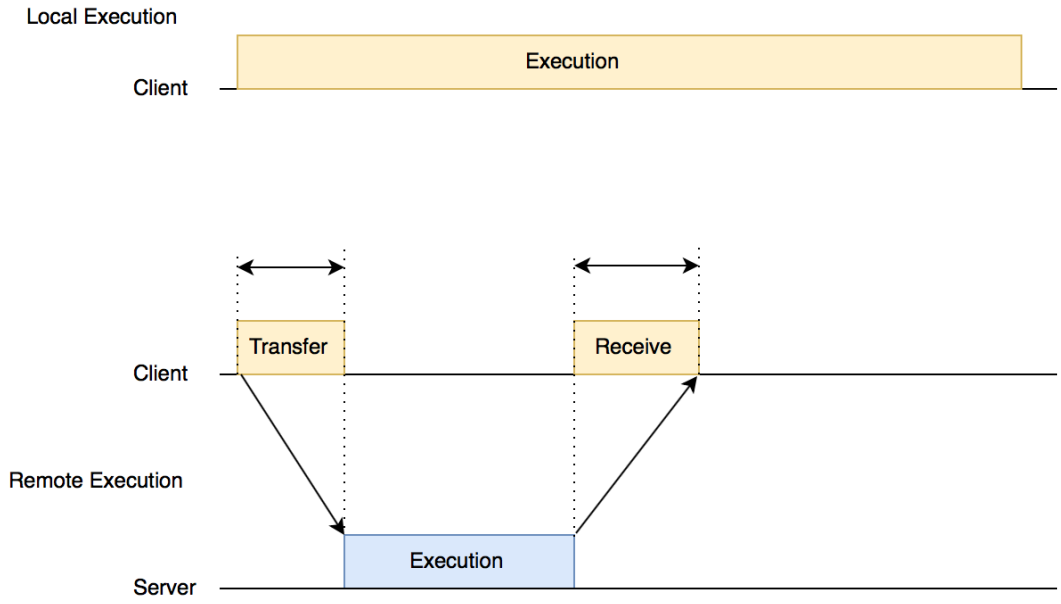


Figure 1.1: Offloading a task to a Remote Server

To be able to run such applications, the device needs an almost constant connection to a remote server. This connection can be through a telecommunication network or through a WiFi Access Point. But constant communication with a remote server causes a high bandwidthload and workload on the server [9].

In certain cases communication with the server is not possible. For example, in a foreign country it could cost a lot of money to use roaming in a network. Or in case of a major accident or a natural disaster a server could be overloaded, not reachable at all or not reachable because there is no network at all. But applications that have time constraints or cannot be executed on the client device at all need an auxiliary resource. In these cases it is possible to use nearby resources to offload a task to carry out the calculation. Furthermore, remote servers are utilized by multiple clients and are not dedicated to one client. Nearby resources on the other hand, like smartphones, tablets and laptops, are more likely to be dedicated to one single client. In some cases, this can even result in nearby resources being more favorable than a remote server.

1.2 Background

This section explains the information and techniques used in this thesis.

1.2.1 Literature Review

This section presents some of the recent surveys on mobile cloud computing as a form of computation offloading. So far cloud computing has gained a lot of popularity in the past few years [9]. A few examples are Microsoft Azure, Amazon Elastic Cloud Computing (EC2) and Google Cloud Platform. These platforms provide a network of hardware and software for computation and storage of data, but are mostly thought to be for non-mobile devices. [15, 9] describe several definitions for mobile cloud computing systems in contrast to a common cloud computing interface. While cloud computing is not very energy aware, mobile cloud computing interfaces mainly aim at improving a mobile system in terms of energy. This means that inputs of tasks are possibly pre-processed to be as small as possible to the effect that overhead of communication is as low as possible.

1.2.2 Computation Offloading

Computation offloading is a technique of executing an application or tasks on another device. For this process the data, if needed, is sent to the other device, which will then perform the specified operations and return the results to the client. In some cases data is pre-processed to increase the efficiency of computation offloading. The goal of this procedure is to improve performance and reduce energy consumption. [9, 8]

1.3 Structure of Thesis

This thesis begins with introducing to the state of art of using auxiliary resources for offloading parts of applications and explaining the basic challenges posed by this model. It then continues with presenting a way to handle some of these challenges. Chapter 3 begins with explaining the setup of the experiment and benchmarks used. The last chapter presents and discusses the results measured in the experiment and then evaluates the results in terms of energy and time.

2 Auxiliary Resources

Auxiliary resources are resources that can be used to offload parts of applications to a different resource in order to save energy and/or overcome computational limitations on embedded systems, e.g. through Mobile Cloud Computing. But as mentioned in Chapter 1, in the absence of usable remote servers nearby mobile embedded systems are possibly available as auxiliary resources. This chapter explains the concept of offloading applications to a remote server and then propose a model of nearby resources.

2.1 System Model

The original system for computation offloading consists of a client and a remote server where the client is connected to a server through a network. The client sends task input data to a remote server and receives the result after the server has finished computation as shown in figure 2.1. This is done by a simple Client-Server communication.

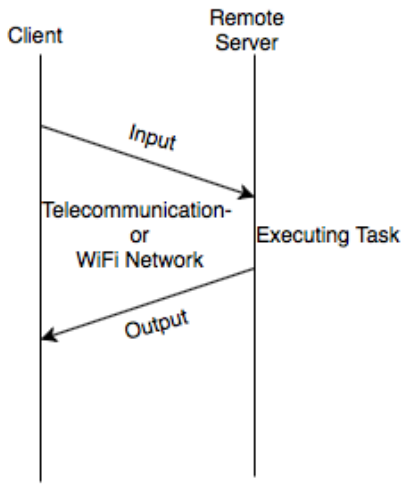


Figure 2.1: Offloading System Model

In this model we introduce the middleware server as a possible substitute for a remote server. Middleware servers in this model basically act as remote servers. The difference is the way of communication. While using Wi-Fi the middleware acts as an access point and accepts a direct connection from the client. It can then either execute the task or offload it

further to the remote server. Bluetooth can also be used to communicate with the client. While Bluetooth is less power consuming, its low bandwidth makes it unsuitable for the transfer of larger files.
Another question is whether or not offloading is favorable in terms of time or energy. Ideally it improves both.

2.1.1 Task Model

In the following system model the client will be referred to as CL, the middleware server as MS and the remote server as RS. $CL >> RS$ will describe the offloading of a task from a client to a remote server in analogy to middleware-client and middleware-remote server offloading.
Given is a set T of n independent tasks, where each task $\tau_i \in T$ is characterized by following average timing parameters:

- C_i: Local execution time on the client for a task τ_i.
- R_i^{device}: Response time of the task on the remote server. The time interval from the arrival of the input on "device" until CL starts receiving the output for a task τ_i.
- $\overset{[source]\rightarrow[destination]}{\triangle t_i}$: Transfer time of a task τ_i where source and destination $\in$ {CL, MS, RS}

2.1.2 Power Model

Power consumption in this thesis is always considered for the client only.

- $P_{state(comp)}^{component}$: Power consumption of the client while a certain component is in a specific state. Where *component* $\in$ {CPU, NIC, BT}, *state*(CPU) $\in$ {idle, run} and *state*(NIC, BT) $\in$ {idle, off, trans, rec}

2.1.3 Energy Model

Energy consumption is only considered for a client. A client can execute the task independently (figure 2.2a) which results in an energy consumption of

$$\overset{CL}{E_i} = C_i * P_{run}^{CPU}.$$

The total energy consumption of the client offloading a task to the middleware server, figure 2.2c:

$$\overset{CL>>MS}{E_i} = \overset{CL\rightarrow MS}{\triangle t_i} * P_T + R_i^{MS} * P_{idle}^{CPU} + \overset{MS\rightarrow CL}{\triangle t_i} * P_R.$$

Executing a task on a remote server, figure 2.2b:

$$E_i^{CL>>RS} = \triangle t_i^{CL \rightarrow RS} * P_T + R_i^{RS} * P_{idle}^{CPU} + \triangle t_i^{RS \rightarrow CL} * P_R.$$

Using the middleware to offload a task, figure 2.2c to the remote server results in a total energy consumption of:

$$E_i^{CL>>MS>>RS} = \triangle t_i^{CL \rightarrow MS} * P_T + (\triangle t_i^{MS \rightarrow RS} + R_i^{RS} + \triangle t_i^{RS \rightarrow MS}) * P_{idle}^{CPU} + \triangle t_i^{MS \rightarrow CL} * P_R.$$

Where $P_T = P_{run}^{CPU} + P_{trans}^{NIC}$ and $P_R = P_{run}^{CPU} + P_{rec}^{NIC}$. Based on these equations offloading a task to a remote server is beneficial in terms of energy if $E_i^{CL>>RS} < E_i^{CL}$ and an improvement in time can be achieved if $\triangle t_i^{CL \rightarrow RS} + R_i^{RS} + \triangle t_i^{RS \rightarrow CL} < C_i$. Analogical for the middleware server. If offloading is possible, an interface can decide based on these equations if the task is executed locally, remotely on the middleware server or on the remote server.

2.2 Server Utilization

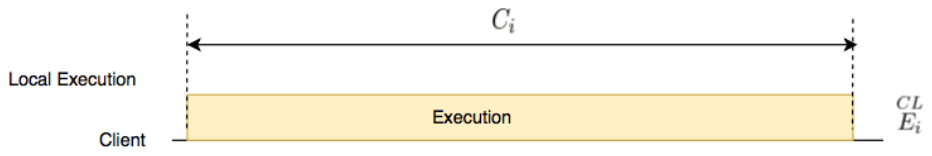
As a remote server may not serve just one but multiple clients, its response time may vary depending on the number of clients served. The server should reserve a specific amount of resources to guarantee a certain response time [16]. In addition to server utilization, the bandwidth in a wireless network can vary depending on different factors. Low bandwidth can result in communication becoming the leading overhead [14].

Guaranteed response time gR_i can be calculated by multiplying the initial response time R_i by the number of clients N utilizing the server.

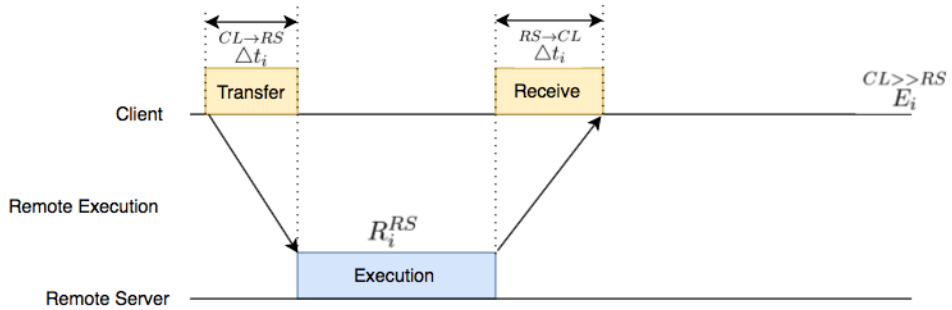
$$gR_i = R_i * N$$

A server utilized by 3 clients for example would reserve 1/3 of its resources for each client. That means the server would guarantee a response time of the initial response time for serving one client multiplied by 3. This means for an initial response time of 1 ms the server would guarantee a response time of 3 ms as shown in figure 2.3.

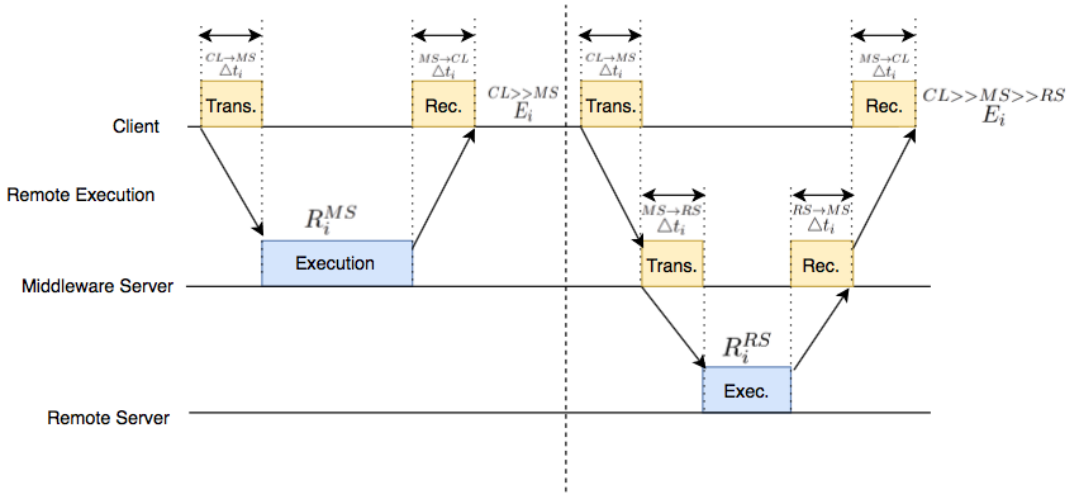
Nearby resources therefore can be chosen to be dedicated to one client, that can result in middleware servers having an even better response time than a more powerful remote server.



(a) Local execution



(b) Remote execution on a remote server



(c) Remote execution using a middleware server

Figure 2.2: Local execution and forms of offloading

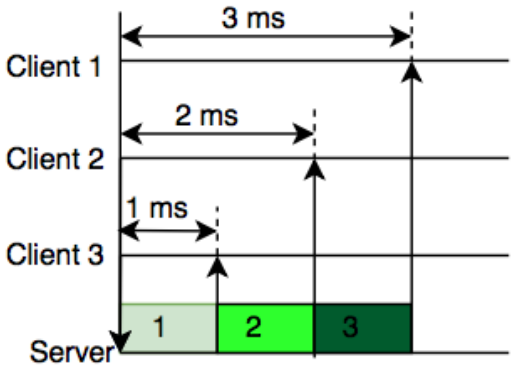


Figure 2.3: Server utilization with 3 clients for 1 ms tasks

2.3 Challenges

Using a remote server requires a constant connection to a network. Maintaining this connection can be challenging especially if the client is moving. Furthermore, the bandwidth may vary and the connection may be disrupted, causing the whole process to restart [8]. Although these problems can be handled by different applications [17], these challenges still reduce the efficiency of the offloading process. A highly utilized server can also cause delays which may lead to tasks missing deadlines or losing the benefit in terms of energy. Therefore we introduced the middleware server in this thesis.

Using a middleware server as a substitute for the remote server can counteract some of those challenges and in some cases may also improve the efficiency of offloading. Middleware servers can basically be any nearby device that supports the application. We assume that the middleware server is not utilized by different clients and is more powerful than the client making use of it. As mentioned above, the middleware server can also be used to as a connection to the remote server, for example in cases where the client only supports Bluetooth.

3 Design and Implementation

This chapter describes the setup of the experiment done to measure power consumption of the device and execution time of the tasks. At first the hardware used for this experiment is explained, followed by an introduction into test design and benchmarks.

3.1 Setup

For the experimental evaluation in this thesis, a total of 3 ODROID XU4s were used. One of them was configured as client, one as the middleware server and one was used to monitor and log the power consumption of the client. We assume that the devices are already connected when starting an offloading process.
The client was specially configured to represent a device with very limited resources and therefore low computational power. Client and middleware Server are connected either through a direct WiFi connection or Bluetooth. For transferring files between the middleware and the client a client-server C-Program was used.

3.1.1 ODROID XU4

The ODROID XU4 is a Heterogeneous Multi-Processing Unit, running on ARM big.LITTLE CPU. More specifically it is running on Samsung Exynos5422 CPU which consists of Cortex A15 and Cortex A7 CPUs and has 2GB of RAM. The Cortex A15 provides four big cores with up to 2GHz and the Cortex A7 provides four LITTLE cores with up to 1.4GHz. It can run both Android and Linux [4]. The XU4s run on Ubuntu MATE. Ubuntu MATE uses the Ubuntu OS as a base and adds the MATE desktop which is a desktop environment with an especially user-friendly design [7].

3.1.2 ODROID SmartPower

The ODROID SmartPower is a power meter made by hardkernel. It has a USB interface to communicate with the monitoring device. Through this interface it sends information about voltage, current, power, and energy consumption of the measured device. This data are then logged to a file. The power meter has a sample rate of 10 Hertz and a fault tolerance of 2 % [3].

3.1.3 Client-Configuration

The client device is representing a mobile embedded system with low computational power. To simulate such a device the client is configured to run on one CPU on 3 different frequencies:

- Low - 500 MHz
- Medium - 1 GHz
- High - 1.4 GHz

These configurations are achieved through turning off all but one CPU, by just changing a CPU's online state. Afterwards a simple script [1] is used to lock the frequency of the CPU on to above frequencies.

3.1.4 Middleware Server and Remote Server

The middleware server in this experiment is a non-configured ODROID XU4. In this case it represents a mobile embedded system more powerful than the client. It is running on all 8 CPU cores with the 4 LITTLE cores being clocked at 1.4 GHz and the 4 big ones at 2 GHz.

To represent a resource-rich remote server a laptop with 8GB of RAM, an IntelHD GPU and an Intel Core i5 with 2.6GHz running on Ubuntu was used.

3.1.5 Monitoring

Since the ODROID XU4 has no sensors for its own power consumption, an external device is used to monitor the power consumption of the client. For this experiment an ODROID SmartPower is used to monitor the clients power consumption, because it is capable of monitoring the power consumption via an USB interface and logging everything to a file on different device [3].

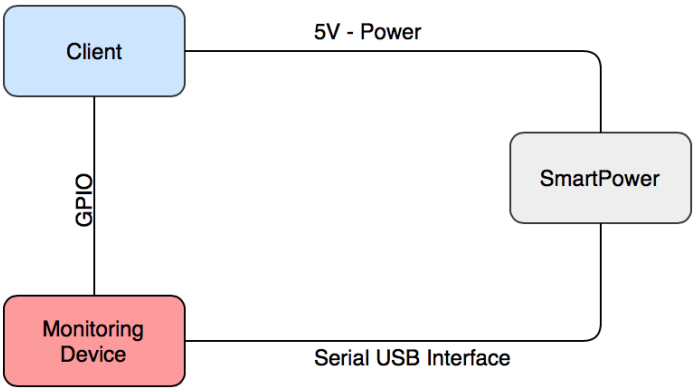


Figure 3.1: Monitoring client

The logging device is another ODROID XU4. It reads the data from the power meter and logs it to a file with the help of a C++ API [13] which reads the serial data provided by the USB interface. In this setup the logging device is connected to the client via GPIO pins. Through the GPIO pins the client signals what to record and when to start and stop. The client signals the start of a certain process, for example the start of a file transfer benchmark, and then signals the exact moments when the file transfer starts and ends. In this way the power consumption is very precisely recorded and can be logged automatically through a script.

Only if the defined GPIO pin is on high, the program actually writes the data to the file. This way it is ensured that the collected data refer exclusively to the process signaled before. As seen in Listing 3.1 the signal turns off immediately after the execution of a benchmark. Those signals can also be implemented within any program if only part of the whole benchmark is to be measured.

Listing 3.1: Example GPIO signal

```
#!/bin/bash
...
gpio mode 7 out

#Signaling the start of a program
gpio write 7 1

./prime99k

#Signaling the end of said program
```

```
gpio write 7 0
...
```

3.2 Test Design

Every benchmark was run 100 times in each configuration to take an average of execution time on each device (client, middleware, remote server). The bodytrack application was modified to output only the coordinates of a tracked body and not the whole image. Facedetect was tested with one image of a face and was modified to output an image with marked face and eyes. Squeezenet accepts and classifies an image and outputs a text file with the determined class. The inputs and outputs of each offloaded task can be seen in Table 3.1.

Table 3.1: Benchmarks

τ_i	Task	Input Size	Output Size
τ_1	Bodytrack Small	2.5 MB	289 B
τ_2	Bodytrack Medium	5 MB	573 B
τ_4	Prime Numbers N=45.000	6 B	5 B
τ_3	Prime Numbers N=99.000	6 B	5 B
τ_5	Arbitrary Matrix Operations 300x300	1.4 MB	720.1 kB
τ_6	Facedetection	6.7 kB	21.3 kB
τ_7	Squeezenet	54.3 kB	11 B

A script then runs every benchmark 100 times, changes the frequency and executes every benchmark again for each configuration.

3.2.1 Bandwidth

Bandwidth was analyzed by sending and receiving 10 MB packages from client to middleware and remote server to determine transfer times. Sending and receiving for the middleware server was done with a simple client-server C program. The server accepts a connection from the client and stays connected during the transfers. For file transfer in this thesis C's sendfile-function was used as shown in Listing 3.2.
In this thesis one remote server in Berlin was used to test the bandwidth between the client and a remote server and another one was tested in Karlsruhe, by using a speedtest [6]. Unlike the connection with the middleware, the client connected to the remote server through a ftp connection.

Listing 3.2: C-Sending

```
clock_gettime(CLOCK_MONOTONIC, &tstart);
```

```
while (((sent_bytes = sendfile(serverfd, fd, &offset, BUFSIZ)) > 0)
                && remaining_data > 0) {
        remaining_data -= sent_bytes;
}

clock_gettime(CLOCK_MONOTONIC, &tend);
```

3.2.2 Benchmarks

The benchmarks selected for this evaluation, are chosen to have different input and output sizes. Some benchmarks were reconfigured to generate a smaller output and/or have a smaller input set. They are also chosen to have different execution times and varying workloads.

PARSEC Benchmark Suite: Bodytrack

"The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors." [5] PARSEC provides a wide range of benchmarks with different workloads and inputs/outputs.
Bodytrack is a computer vision application that tracks a 3D pose of a human body with multiple cameras. The used inputs on this benchmark were "simsmall" and "simmedium" with simsmall using 4 cameras, 1 frame, 1000 particles and 5 annealing layers and simmedium 2 frames and 2000 particles [11]. Bodytrack was used in the evaluation because computer vision applications are commonly used in a lot of mobile embedded systems.

Calculating N Prime Numbers

This benchmark is a simple mathematical calculation of the first N prime numbers. The input in this benchmark is the number N for the first N-Prime Numbers and the output is the count of the first N prime numbers. Calculating prime numbers was chosen in this evaluation to represent a heavy mathematical calculation.

Arbitrary Matrix Operations

Common operations in modern programs are mathematical operations on matrices [9]. Therefore this benchmark performs a few arbitrary mathematical operations on two large matrices. It is written in python and using the numpy package for storing data and

performing mathematical operations. The input are two npy-files, each storing a matrix, and the output is one npy-file storing the resulting matrix.

OpenCV Facedetect

Facedetect is an example application in the openCV library. This benchmark accepts an image as an input or uses a camera to read an image sequence and marking the position of a face and eyes. It was chosen as it is a commonly used feature in mobile embedded systems with cameras.

deepRacin squeezenet

Squeezenet is an architecture of a Convolutional Neural Network. It solves a 1000-class classification problem of ImageNet-data and was developed to use little space and be fast, still delivering good classifications [2].

4 Results and Discussion

This Chapter presents the results of the measuring which will be evaluated in terms of energy and time and shows whether or not offloading is favorable for the tasks mentioned in table 3.1. Further it discusses the effect of varying bandwidths to the remote server and the effects of server utilization.

4.1 Power Consumption

Since the XU4 does not have sensors to measure each component, like CPU and NIC[1], the power model for this evaluation was separately updated to consider the power consumption of the whole device. In this evaluation the power consumption of one component in a certain state should be seen as the power consumption for the whole device while that component is in a certain state. Also when CPU was measured, NIC and Bluetooth were turned off, when NIC was measured, Bluetooth was turned off and when Bluetooth was measured NIC was turned off.

Figure 4.1 shows the power consumption for each component in different states for each configuration. CPU(idle) is also the power consumption for BT(off) and NIC(off). "NIC DIRECT" is the power consumption during a direct connection between client and middleware. The direct connection consumes less power than a connection through a network, especially when in the idle state. As expected Bluetooth has the lowest power consumption. But in general the idle power consumption does not vary as much as running, transmission or receiving states of the components, for the different configurations.

[1]Network Interface Card

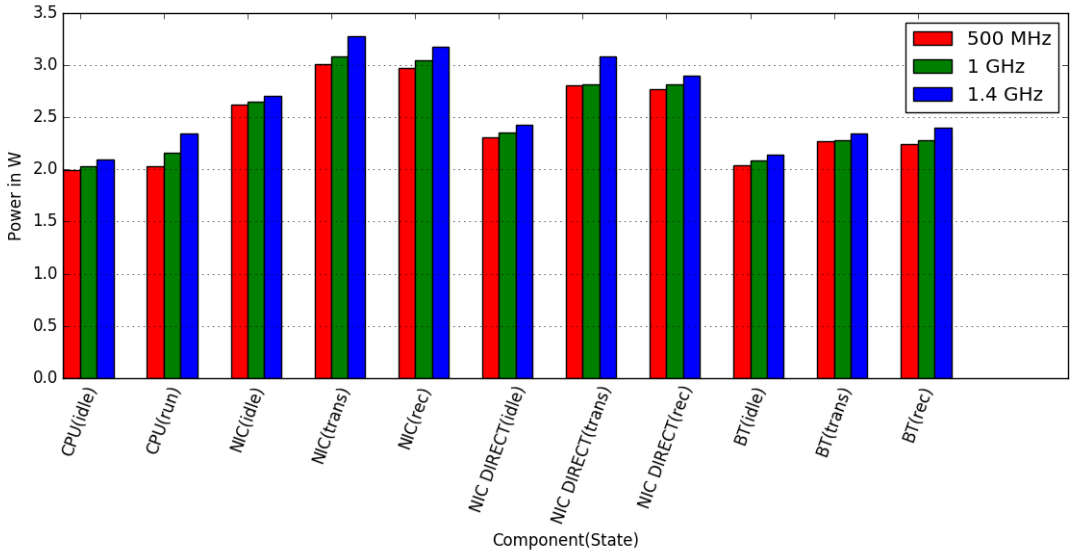
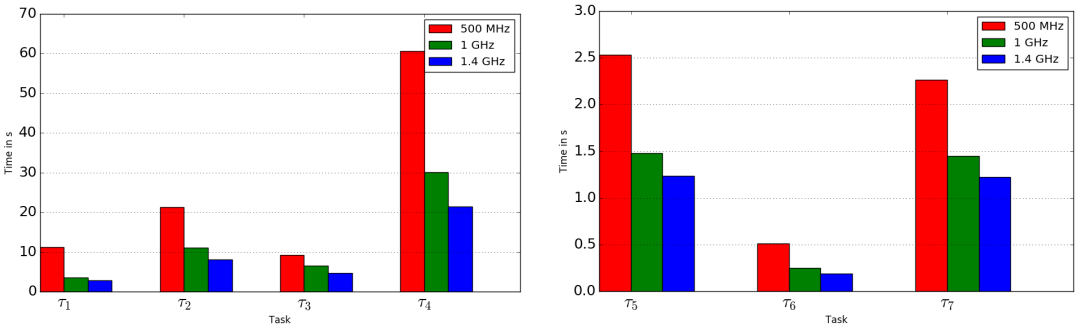


Figure 4.1: Power Consumption of the XU4 for all different configurations

4.2 Execution Time

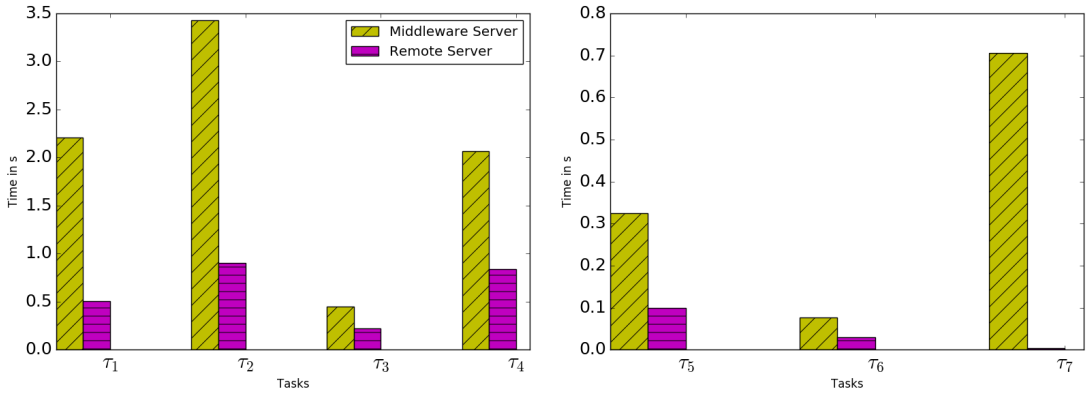
This section shows the results of the task evaluation, showing local execution times and response times. Figure 4.2 shows the execution times of all applications tested in the experiment. Where 4.2a shows tasks τ_1 - τ_4 and 4.2b shows tasks τ_5 - τ_7. Execution times vary from 0.5 seconds up to 60 seconds. Tasks like τ_4 that have an input as small as 5 Byte and an execution time of up to 60 seconds, are good candidates for effective offloading. But also other tasks, with bigger input like bodytrack can save time and energy using computation offloading. The much improved execution time suggests that energy consumption for local execution should be much lower for a higher frequency.



(a) Local execution time Tasks $\tau_1 - \tau_4$ **(b)** Local execution time Tasks $\tau_5 - \tau_7$

Figure 4.2: Local execution times on client in all configurations

Response times are an important aspect of computation offloading. The faster the server executes the task and starts sending the output, the better the effect of offloading. A server's response time varies, depending on specifications and utilization of the server. Figure 4.3a and 4.3b show response times of all tasks. All response times are much lower than clients execution times and remote server response times are lower than those of middleware servers. Task τ_7 is executed on a different remote server (Ubuntu, Nvidia GTX 1080), since this tasks works mainly on the GPU. This tasks shows a very big difference in execution times, ranging from about 2.5 seconds, for local execution, 0.7 seconds, on the middleware server, and down to 0.004 seconds, on the remote server.

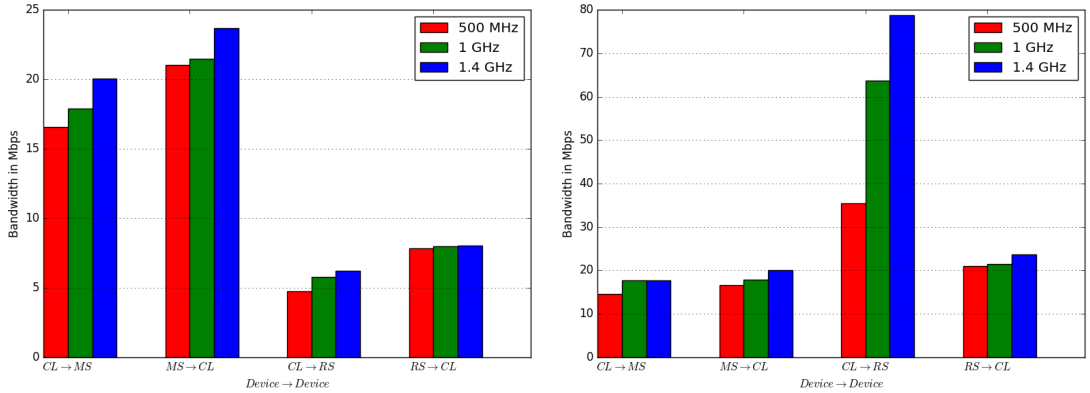


(a) Response time Tasks $\tau_1 - \tau_4$ **(b)** Response time Tasks $\tau_5 - \tau_7$

Figure 4.3: Response time middlware and remote server

4.3 Bandwidth

In order to estimate transfer times, the different bandwidths between the devices have been measured. Figure 4.4 shows the bandwidths measured in this experiment. As expected, bandwidth between the client, middleware and remote server varies depending on the configuration of the client and the chosen remote server, it can be observed that the bandwidth between client and middleware server over a direct Wi-Fi connection is much higher than between client and the remote server in Berlin(Figure 4.4a). A remote server that is in closer vicinity and has a higher bandwidth however surpasses the bandwidth between middleware and client(Figure 4.4b). Also the download speed it is in general a little higher than the upload speed. In the lowest configuration upload speed is more than 3 times as high between client and middleware server than between client and remote server, resulting in longer transfer times for $CL >> RS$. Additionally to the lower bandwidth a network connection also consumes more power than a direct connection as can be seen in 4.1.



(a) Bandwidth MS, RS_{Berlin} **(b)** Bandwidth MS, $RS_{Karslruhe}$

Figure 4.4: Bandwidth Client

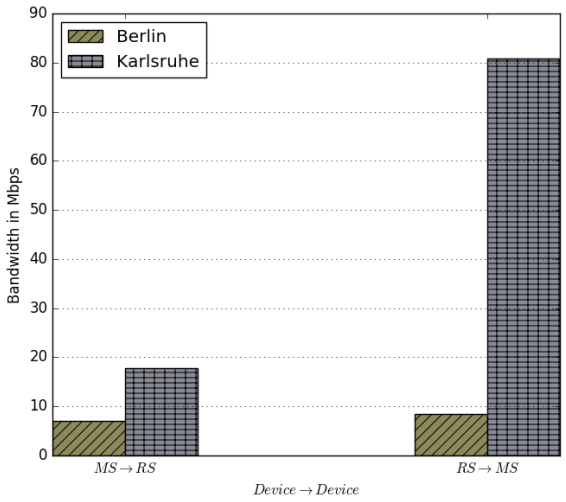


Figure 4.5: Bandwidth between MS and RS

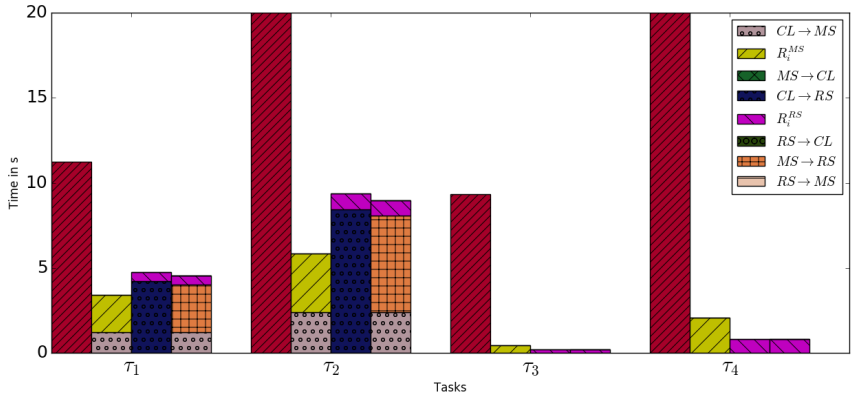
4.4 Time Consumption

The time consumption for the local execution is, as before mentioned C_i. Total time for offloading cases, $O_i^{A>>B}$, for this evaluation is defined as:

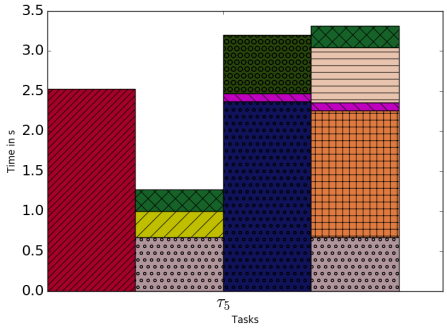
- $O_i^{CL>>MS} = \overset{CL \to MS}{\triangle t_i} + R_i^{MS} + \overset{MS \to CL}{\triangle t_i}$
- $O_i^{CL>>RS} = \overset{CL \to RS}{\triangle t_i} + R_i^{RS} + \overset{RS \to CL}{\triangle t_i}$
- $O_i^{CL>>MS>>RS} = \overset{CL \to MS}{\triangle t_i} + \overset{MS \to RS}{\triangle t_i} + R_i^{RS} + \overset{RS \to MS}{\triangle t_i} + \overset{MS \to CL}{\triangle t_i}$

Since the main part in this section refers to total time consumption, offloading to the middleware server is always done by WiFi, since it has a higher bandwidth. Most tasks

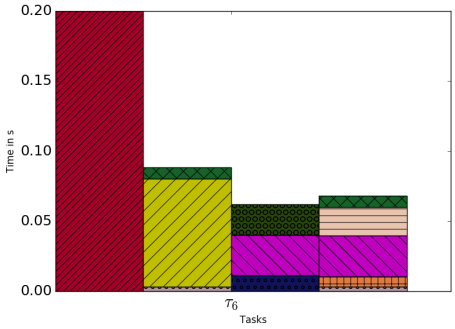
used in the evaluation have very small inputs and outputs so that the transmission time is not visible for every task and direction. Figure 4.6 shows local execution times, for a client in low configuration, compared to the 3 offloading types introduced in the model. Where $A \rightarrow B$ means the transmission time from device A to device B for a given task. From left to right it is local execution, remote execution on middleware server, remote execution on remote server and remote execution on a remote server using the middleware. Task τ_5 is the only task where local execution times is not the highest of all. Mostly the main part of offloading is the response time R_i^{device}. In some cases it is even the only relevant part. Although the remote server's response time is much faster than that of the middleware servers, task τ_1, τ_2 and τ_5 are faster when offloaded to the middleware server. These tasks are also the tasks with highest inputs/outputs size. Due to the higher bandwidth between middleware and client, transmission times for these tasks are much lower.



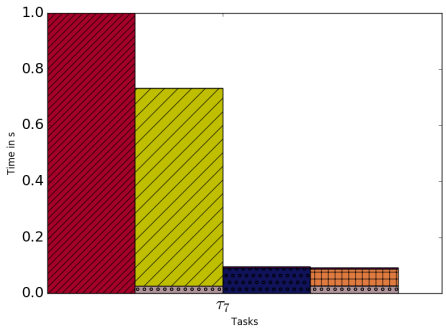
(a) Time consumption $\tau_1 - \tau_4$



(b) Time consumption τ_5



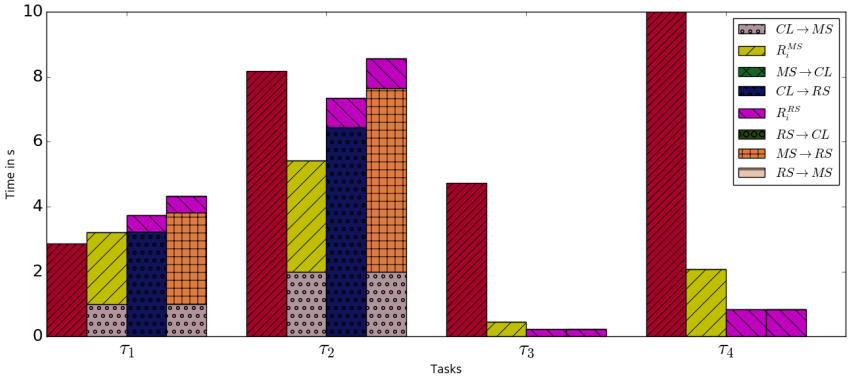
(c) Time consumption τ_6



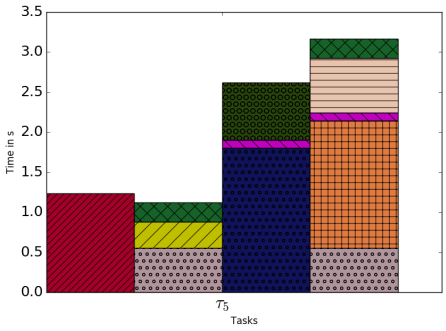
(d) Time consumption τ_7

Figure 4.6: Time consumption local execution (lowest configuration) vs. computation offloading (RS_{Berlin})

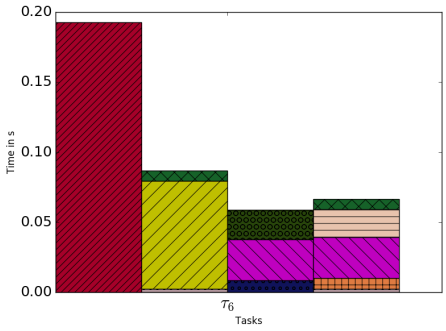
Figure 4.7 compares local execution times to offloading, when the client is configured in a higher configuration than above. In this state the local execution on the client is more favorable for some tasks like τ_1. But even with this configuration it can be observed that for example face detection (Figure 4.7c) takes only 50% of the time, when offloaded to the middleware.



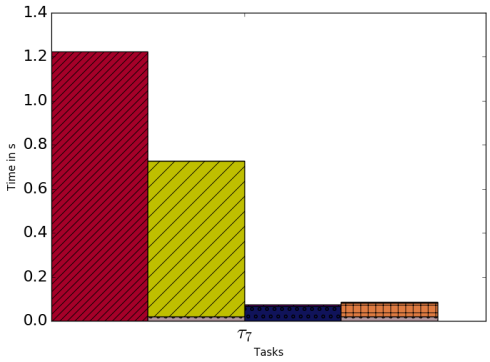
(a) Time consumption $\tau_1 - \tau_4$



(b) Time consumption τ_5



(c) Time consumption τ_6



(d) Time consumption τ_7

Figure 4.7: Time consumption local execution (highest configuration) vs. computation offloading (RS_{Berlin})

4.5 Energy Consumption

The energy model in this experiment was adapted to the experiment due to missing sensors for single components in the XU4.

$$\overset{CL}{E_i} = C_i * P_{run}^{CPU}$$

$$\overset{CL>>RS}{E_i} = \overset{CL \rightarrow RS}{\triangle t_i} * P_{trans}^{NIC} + R_i^{RS} * P_{idle}^{NIC} + \overset{RS \rightarrow CL}{\triangle t_i} * P_{rec}^{NIC}$$

$$E_i^{CL>>MS>>RS} = \triangle t_i^{CL\rightarrow MS} * P_{trans}^{NIC} + (\triangle t_i^{MS\rightarrow RS} + R_i^{RS} + \triangle t_i^{RS\rightarrow MS}) * P_{idle}^{NIC} + \triangle t_i^{MS\rightarrow CL} * P_{rec}^{NIC}$$

The device for transmission between client and middleware server was chosen with a view to the total energy consumption for using that device:

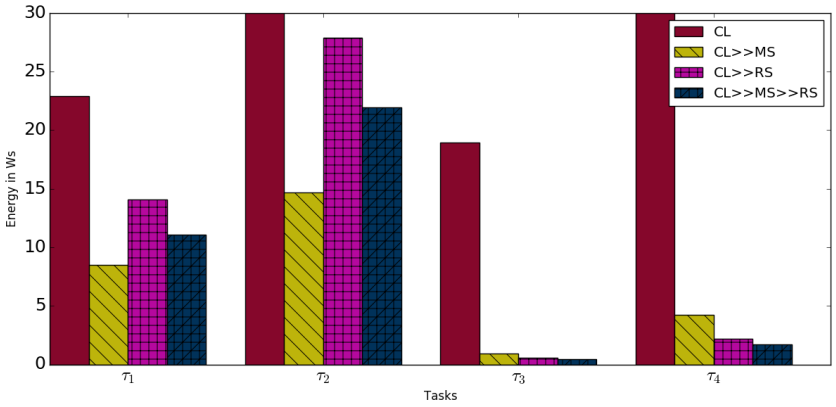
$$E_i^{CL>>MS} = \min \begin{cases} \triangle t_i^{CL\rightarrow MS} * P_{trans}^{BT} + R_i^{MS} * P_{idle}^{BT} + \triangle t_i^{MS\rightarrow CL} * P_{rec}^{BT} \\ \triangle t_i^{CL\rightarrow MS} * P_{trans}^{NICDirect} + R_i^{MS} * P_{idle}^{NICDirect} + \triangle t_i^{MS\rightarrow CL} * P_{rec}^{NICDirect} \end{cases}$$

Analogously the same decision was made for offloading to a remote server through a middleware server. In following figures $CL >> MS$ and $CL >> MS >> RS$ were chosen by lower energy consumption.

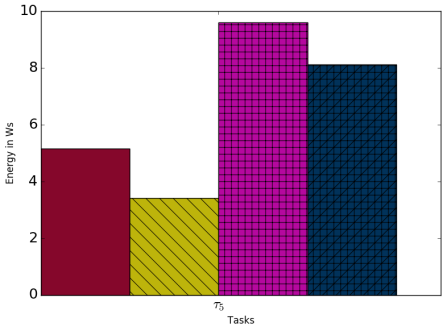
Figure 4.8 shows energy consumption of the client, in low configuration, for each task and offloading type compared to local execution. For offloading to the middleware server the better option between using Bluetooth and Wi-Fi was chosen. Bluetooth is more efficient especially for very small inputs and outputs. In this experiment it was for τ_3 and τ_4 where Bluetooth was more energy saving than WiFi.

Tasks τ_1, τ_2 and τ_5(Figure 4.8a and 4.8b) save the most energy using a middleware server, due to big input and output. Server utilization will cause the middleware to be even more efficient, but a higher bandwidth of the remote server may change the effect of offloading. For task τ_7(Figure 4.8d), the use of a remote server reduces energy consumption by 99.4% compared to local execution and by 86.3% compared to offloading to a middleware server. But a higher utilized server and a low bandwidth can change the advantage of a remote server even for a so much improved task compared to middleware.

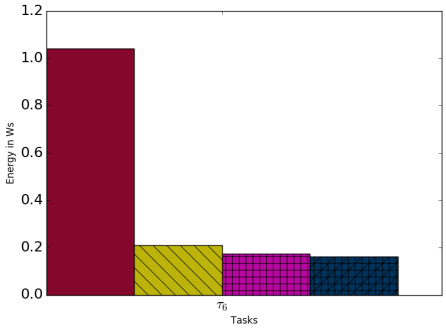
Generally every task can be improved in terms of energy consumption using remote servers. Considering the measured bandwidths in this experiment, using nearby resources is even more favorable than using a remote server either remotely executing them on the middleware or using the middleware to offload it to the remote server. This is due to the much higher bandwidth of the direct connection between the client and middleware server which saves time int the transfer of inputs and outputs.



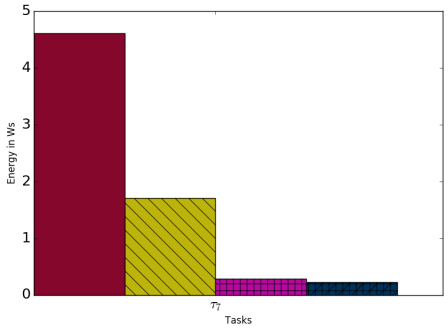
(a) Energy consumption $\tau_1 - \tau_4$



(b) Energy consumption τ_5



(c) Energy consumption τ_6

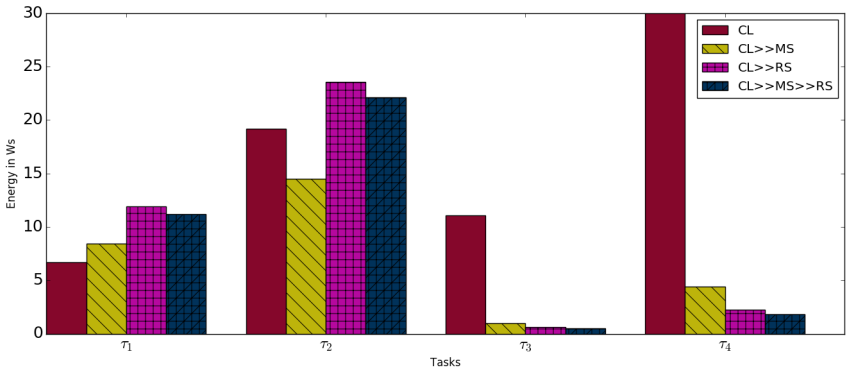


(d) Energy consumption τ_7

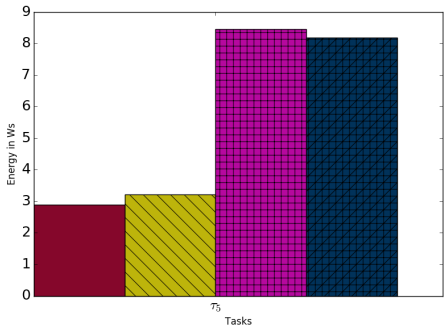
Figure 4.8: Energy consumption local execution (lowest configuration) vs. computation offloading (RS_{Berlin})

On the other hand if the client has more resources, like the highest configuration in the experiment, some tasks are not favorable, in terms of energy, for offloading. This is due to much better local execution time and therefore lower energy consumption. Although power consumption for file transmission also rises, the energy consumption for offloading stays nearly the same because of a higher bandwidth that comes along with higher CPU frequency.

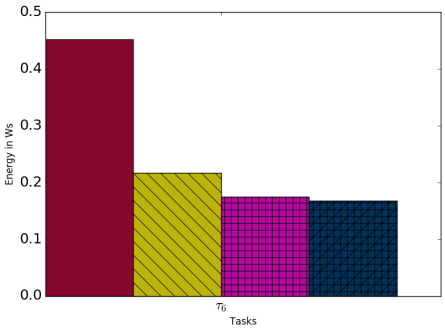
Figure 4.9 shows energy consumption for a client configured with a higher frequency. It can be observed that task τ_1 and τ_5 (Figure 4.9a and 4.9b) are more energy saving if ran on the client than when offloaded. The other tasks still consume less energy when offloaded to a remote or middleware server.



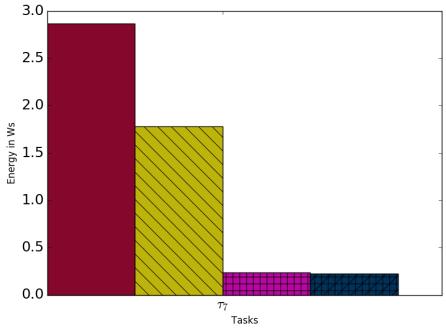
(a) Energy consumption $\tau_1 - \tau_4$



(b) Energy consumption τ_5



(c) Energy consumption τ_6



(d) Energy consumption τ_7

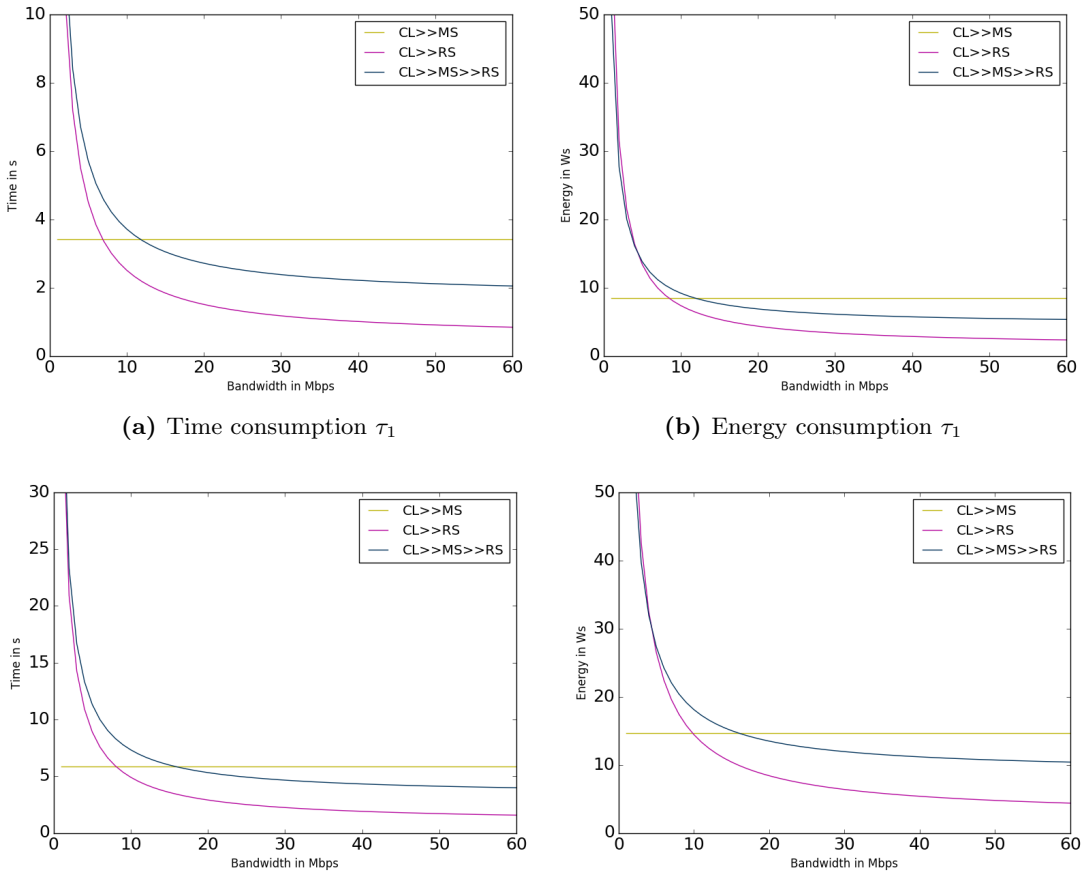
Figure 4.9: Energy consumption local execution (highest configuration) vs. computation offloading (RS_{Berlin})

4.6 Bandwidth Variation

Bandwidth varies depending on the server used and the location of the client. This section shows the effect on offloading time and energy consumption for a range of bandwidths for

every task. The bandwidth between CL and MS was fixed at the value measured in the experiment, while the bandwidth between CL and RS and MS and RS was ranged between 1 and 60 Mbps. Figure 4.10 and 4.11 show how bandwidth affects the quality of effect for tasks where remote servers were less efficient when using a remote server compared to a middleware with a fixed bandwidth. For nearly every task the remote servers surpass the middleware before reaching 10 Mbps in terms of time and energy by about half the bandwidth of the middleware used in the experiment. Task τ_5 is the only task in this experiment where the remote server surpasses the middleware server just at above 15 Mbps.

It is also interesting to note that in all cases in which the middleware was used for offloading to a remote server, more energy was very early consumed than with direct offloading to a remote server. Most of the times only bandwidths below 5 Mbps allow $CL >> MS >> RS$ to be useful.



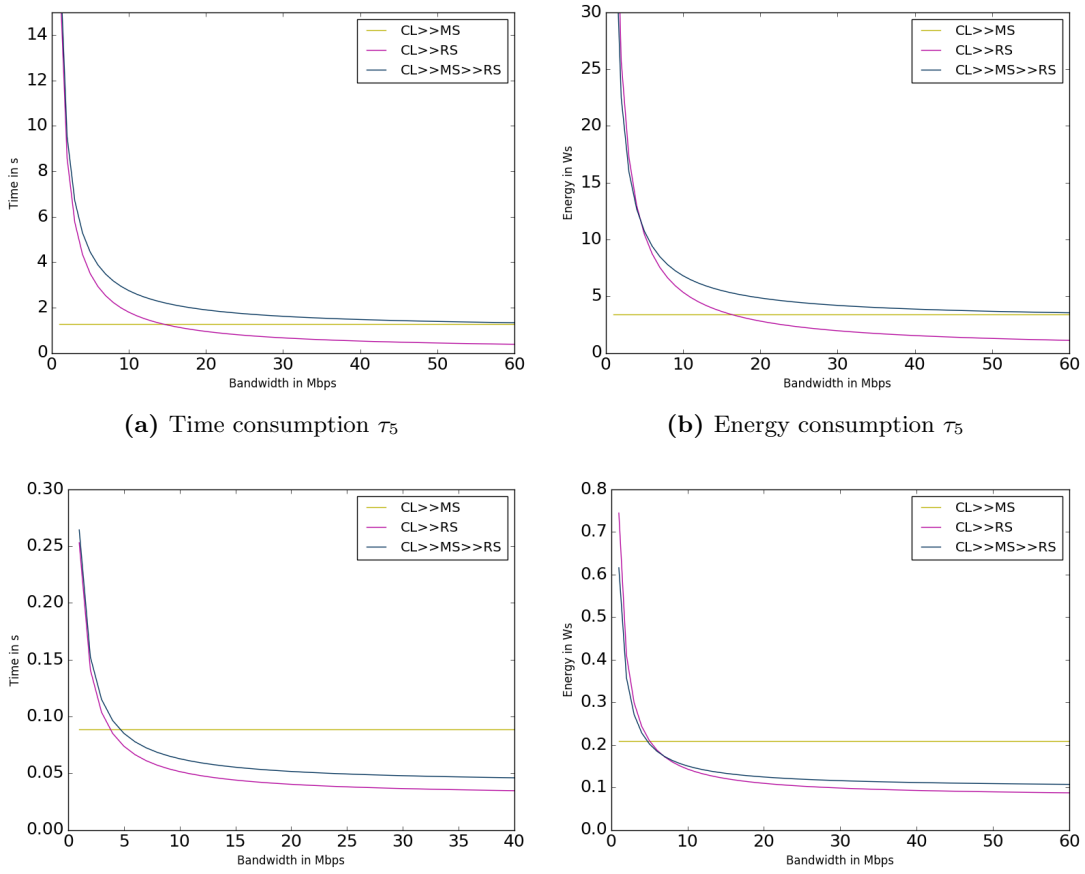
(a) Time consumption τ_1

(b) Energy consumption τ_1

(c) Time consumption τ_2

(d) Energy consumption τ_2

Figure 4.10: Time and energy consumption with applied bandwidth variation between RS and CL and RS and MS (τ_1 and τ_2)



(a) Time consumption τ_5

(b) Energy consumption τ_5

(c) Time consumption τ_6

(d) Energy consumption τ_6

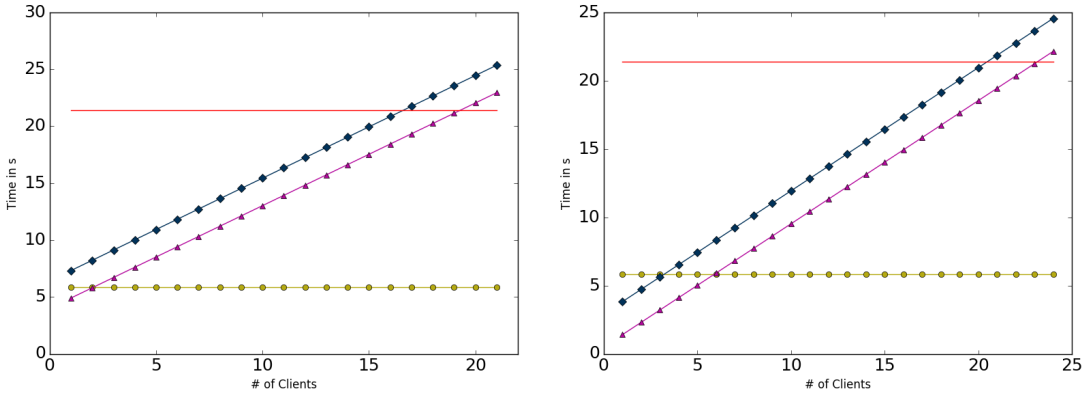
Figure 4.11: Time and energy consumption with applied bandwidth variation between RS and CL and RS and MS (τ_5 and τ_6)

4.7 Remote Server Utilization

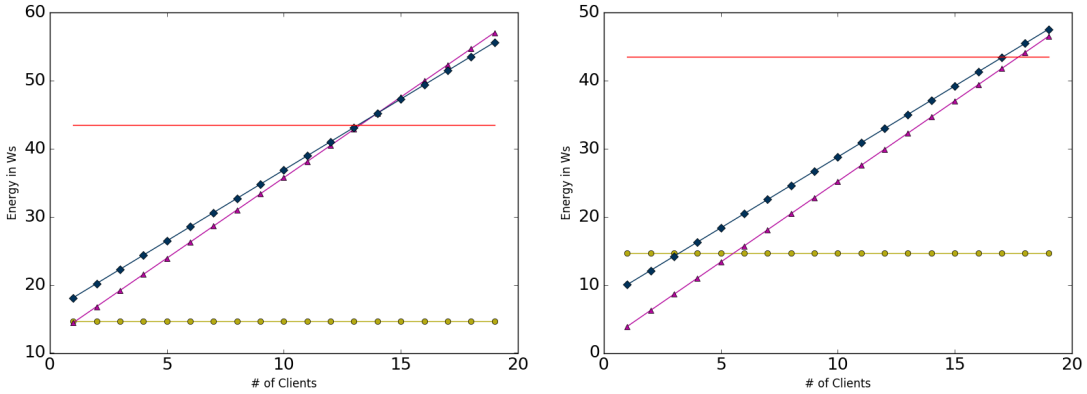
Remote servers are rarely dedicated to just one client. In this section server utilization is applied to $CL >> RS$ and $CL >> MS >> RS$ in order to determine the number of clients utilizing one server where the remote server becomes unsuitable. Values for the middleware server are also fixed in this case. Figures 4.12, 4.13 and 4.14 compare offloading execution times with local execution times and $CL >> MS$ for two different bandwidths.

Figure 4.12a shows that at a bandwidth of 10 Mbps $CL >> RS$ exceeds $CL >> MS$ at just 2 clients and CL at 19 clients utilizing the remote server considering overall execution time or offloading time. Whereas $CL >> MS >> RS$ has an overall higher time consumption than $CL >> RS$. With a bandwidth of 80 Mbps (Figure 4.12b) $CL >> RS$ consumes more time than $CL >> MS$ only after 6 or 7 clients utilizing the server.

Figure 4.12c and 4.12d show energy consumption for the same bandwidths as above. For B=10 Mbps $CL >> MS$ consumes less energy, no matter how many clients utilize the remote server. $CL >> RS$ consumes more energy than CL and also $CL >> MS >> RS$ from 14 clients utilizing it. A bandwidth of 80 Mbps shifts the critical utilization, where $CL >> RS$ becomes more energy consuming than $CL >> MS$, to 6 clients. CL consumes less energy than $CL >> RS$ at 19 clients.



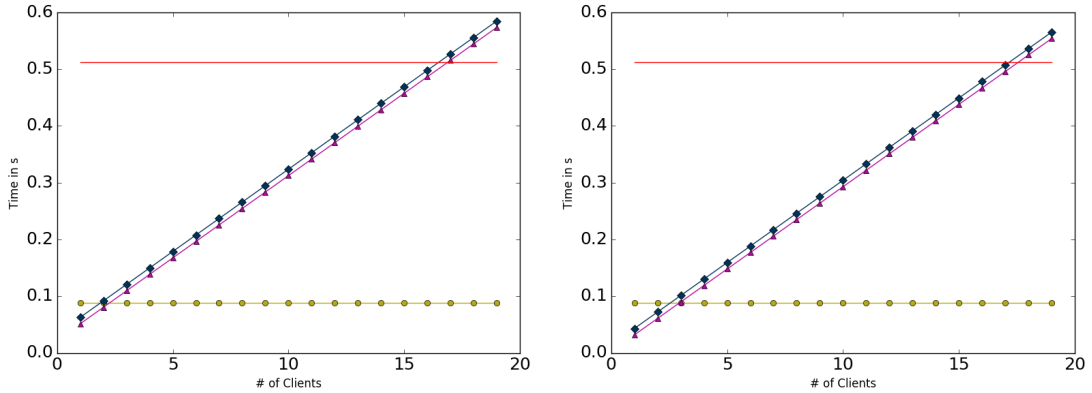
(a) B=10Mbps, Time consumption τ_2 **(b)** B=80Mbps, Time consumption τ_2



(c) B=10Mbps, Energy consumption τ_2 **(d)** B=80Mbps, Energy consumption τ_2

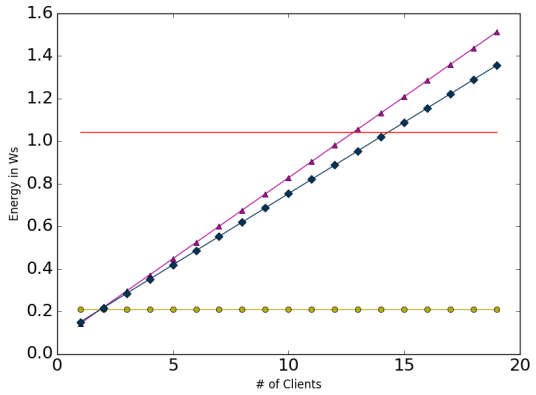
Figure 4.12: Server Utilization: Task τ_2 time and energy consumption. Bandwidth: 10 and 80 Mbps

In case of task τ_6 energy consumption for $CL >> MS >> RS$, as shown in Figures 4.13c and 4.13d, is generally lower than $CL >> RS$ and exceeds energy consumption for local execution at 13 clients utilizing RS or at 15 clients for $CL >> MS >> RS$ at B=10 Mbps. In terms of time consumption $CL >> MS >> RS$ takes more time than $CL >> RS$ at every point. Both surpass client execution time with 17 clients utilizing RS at 10 Mbps and with 19 clients at 80 Mbps. The middleware, in this case, is more energy saving at 3 clients utilizing the remote server.

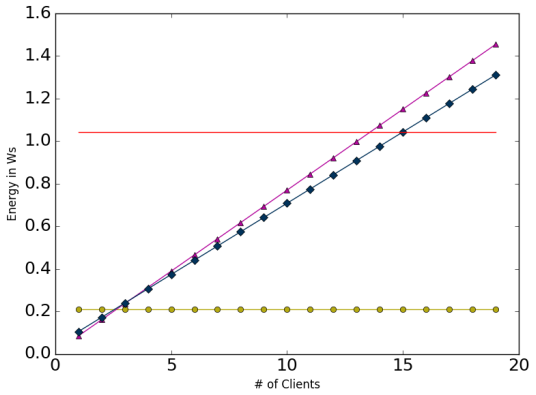


(a) B=10Mbps, Time consumption τ_6

(b) B=80Mbps, Time consumption τ_6



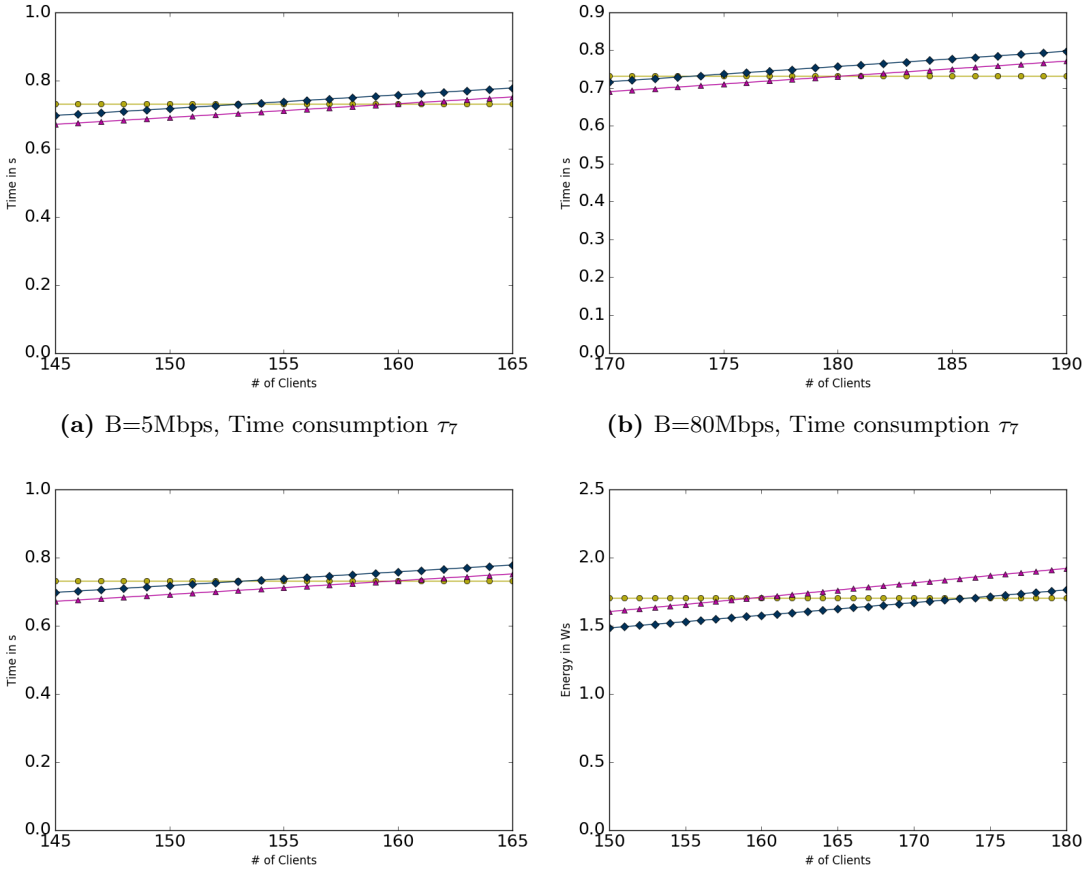
(c) B=10Mbps, Energy consumption τ_6



(d) B=80Mbps, Energy consumption τ_6

Figure 4.13: Server Utilization: Task τ_6 time and energy consumption. Bandwidth: 10 and 80 Mbps

Task τ_7 is a task where remote execution on a remote server is almost always favorable. This is primarily due to the extremely low response time of 4 ms, which is a speedup of 99.5% compared to the response time of the middleware. Only with about 165 clients utilizing the remote server the middleware needs less time than the remote server, with a bandwidth of 5 Mbps (Figure 4.14a), and with about 180 clients, with a bandwidth of 80 Mbps (Figure 4.14b). Energy consumption for $CL >> RS$ exceeds the middleware with about 160 clients utilizing the remote server for both bandwidths, B=5 Mbps (Figure 4.14c) and B=80 Mbps (Figure 4.14d). Local execution would only be more favorable with more than 400 clients utilizing the server.



(a) B=5Mbps, Time consumption τ_7

(b) B=80Mbps, Time consumption τ_7

(c) B=5Mbps, Energy consumption τ_7

(d) B=80Mbps, Energy consumption τ_7

Figure 4.14: Server Utilization: Task τ_7 time and energy consumption. Bandwidth: 5 and 80 Mbps

5 Conclusion

This thesis evaluated the energy consumption of a mobile embedded system with limited resources performing different benchmarks with varying workloads. The client system was monitored in its power and time consumption during different states of the components and for several benchmarks. In addition to standard computation offloading using a remote server, a nearby resource can be used as a middleware server. It has two advantages over a remote server, which are way of communication and availability. In the experiment performed in this thesis, it has shown that a middleware server can also save time and energy. In some cases it even saves more energy than using a remote server, especially on larger inputs and outputs. But also when a server is utilized and the bandwidth between client and server is low, a middleware becomes more favorable than a remote server. For the face detection task a middleware server could save approximately 75% compared to a remote server with a bandwidth of 5 Mbps and a utilization of 10%. In practice that would mean if the client in this experiment would run on a battery with a capacity of 2,000 mAh at 5V, face detect could be executed on 34,538 images, before the client ran out of battery, if executed locally on the client. If offloaded to the middleware it could be ran on 171,939 images.

To make the right offloading decision the client would need to scan the environment for nearby resources and remote servers it could connect to. It could then make a decision based on known data on execution, response and transfer times of the task that is to be executed. Nearby resources can for example be mobile embedded systems that are significantly more powerful than the client.

In conclusion this thesis showed that a client can benefit from offloading tasks to a nearby resource, especially for clients with very limited resources.

List of Figures

List of Source Codes

Bibliography

[1] odroid-cpu-control. https://github.com/mad-ady/odroid-cpu-control, 2016. Last checked: 05.08.2017.

[2] deepracin. https://github.com/mrjel/deepracin, 2017. Last checked: 26.07.2017.

[3] Odroid smartpower. http://odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower, July 2017. Last checked: 14.07.2017.

[4] Odroid-xu4, specifications. http://odroid.com/dokuwiki/doku.php?id=en:xu4_hardware, 2017. Last checked: 09.07.2017.

[5] The parsec benchmark suite. http://parsec.cs.princeton.edu, 2017. Last checked: 12.07.2017.

[6] speedtest-cli. https://github.com/sivel/speedtest-cli, 2017. Last checked: 05.08.2017.

[7] What is ubuntu mate? https://ubuntu-mate.org/what-is-ubuntu-mate/, 2017. Last checked: 09.07.2017.

[8] P. D. Arani Bhattacharya. A survey of adaptation techniques in computation offloading. In *Journal of Network and Computer Applications 78*, pages 97–115, 2017.

[9] S. A. M. S. U. K. Atta ur Rehman Khan, Mazliza Othman. A survey of mobile cloud computing application models. In *IEEE Communcations Surveys Tutorials, Vol. 16, No. 1*, 2014.

[10] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 155–168, New York, NY, USA, 2015. ACM.

[11] J. P. S. Christian Bienia, Sanjeev Kumar and K. Li. The parsec benchmark suite: Characterization and architectural implications. Technical report, 2008.

[12] J. Li, Z. Peng, B. Xiao, and Y. Hua. Make smartphones last a day: Pre-processing based computer vision application offloading. In *Sensing, Communication, and Net-*

working (SECON), 2015 12th Annual IEEE International Conference on, pages 462–470. IEEE, 2015.

[13] T. Mhlbauer. odroid-smartpower-linux. `https://github.com/muehlbau/odroid-smartpower-linux`, 2014. Last checked: 25.07.2017.

[14] Y. Nimmagadda, K. Kumar, Y.-H. Lu, and C. G. Lee. Real-time moving object recognition and tracking using computation offloading. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2449–2455. IEEE, 2010.

[15] W. R. Niroshinie Fernando, Seng W. Loke. Mobile cloud computing: A survey. In *In Future Generation Computer Systems 29*, pages 84–106, 2013.

[16] A. Toma, S. Pagani, J.-J. Chen, W. Karl, and J. Henkel. An energy-efficient middleware for computation offloading in real-time embedded systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 228–237. IEEE, 2016.

[17] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri. Run time application repartitioning in dynamic mobile cloud environments. *IEEE Transactions on Cloud Computing*, 4(3):336–348, 2016.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift