User Experience Driven CPU Frequency Scaling On Mobile Devices Towards Better Energy Efficiency

Volker Seeker

Doctor of Philosophy

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2017

Abstract

With the development of modern smartphones, mobile devices have become ubiquitous in our daily lives. With high processing capabilities and a vast number of applications, users now need them for both business and personal tasks. Unfortunately, battery technology did not scale with the same speed as computational power. Hence, modern smartphone batteries often last for less than a day before they need to be recharged. One of the most power hungry components is the central processing unit (CPU). Multiple techniques are applied to reduce CPU energy consumption. Among them is dynamic voltage and frequency scaling (DVFS). This technique reduces energy consumption by dynamically changing CPU supply voltage depending on the currently running workload. Reducing voltage, however, also makes it necessary to reduce the clock frequency, which can have a significant impact on task performance. Current DVFS algorithms deliver a good user experience, however, as experiments conducted later in this thesis will show, they do not deliver an optimal energy efficiency for an interactive mobile workload. This thesis presents methods and tools to determine where energy can be saved during mobile workload execution when using DVFS. Furthermore, an improved DVFS technique is developed that achieves a higher energy efficiency than the current standard.

One important question when developing a DVFS technique is: How much can you slow down a task to save energy before the negative effect on performance becomes intolerable? The ultimate goal when optimising a mobile system is to provide a high quality of experience (QOE) to the end user. In that context, task slowdowns become intolerable when they have a perceptible effect on QOE. Experiments conducted in this thesis answer this question by identifying workload periods in which performance changes are directly perceptible by the end user and periods where they are imperceptible, namely interaction lags and interaction idle periods. Interaction lags are the time it takes the system to process a user interaction and display a corresponding response. Idle periods are the periods between interactions where the user perceives the system as idle and ready for the next input. By knowing where those periods are and how they are affected by frequency changes, a more energy efficient DVFS governor can be developed.

This thesis begins by introducing a methodology that measures the duration of interaction lags as perceived by the user. It uses them as an indicator to benchmark the quality of experience for a workload execution. A representative benchmark workload

is generated comprising 190 minutes of interactions collected from real users. In conjunction with this QOE benchmark, a DVFS *Oracle* study is conducted. It is able to find a frequency profile for an interactive mobile workload which has the maximum energy savings achievable without a perceptible performance impact on the user. The developed *Oracle* performance profile achieves a QOE which is indistinguishable from always running on the fastest frequency while needing 45% less energy. Furthermore, this *Oracle* is used as a baseline to evaluate how well current mobile frequency governors are performing. It shows that none of these governors perform particularly well and up to 32% energy savings are possible. Equipped with a benchmark and an optimisation baseline, a user perception aware DVFS technique is developed in the second part of this thesis. Initially, a runtime heuristic is introduced which is able to detect interaction lags as the user would perceive them. Using this heuristic, a reinforcement learning driven governor is developed which is able to learn good frequency settings for interaction lag and idle periods based on sample observations. It consumes up to 22% less energy than current standard governors on mobile devices, and maintains a low impact on QOE.

Lay Summary

In recent times modern smartphones became a part of almost everyone's daily life. They are used both at work and at home to organise appointments, answer emails, play games, video chat with your family, observe the stars and much more. To be able to execute applications of different complexity and to provide smooth graphics, mobile phone technology improved much over the last years. Screens became larger and memory capacity increased as well as computational speed. These improvements, however, come with a cost. They need a constant energy supply from the phone's battery. A mobile phone from 15 years ago had a battery lifetime of almost a week. Unfortunately, with improved technology and a broader application selection, modern phones usually last no longer than a day before their batteries need to be recharged.

Since battery technology does not improve as fast as computational technology, researchers focus on reducing battery consumption by writing software which uses mobile components in a more energy efficient way. This thesis looks at the phone's processor in particular. It is at the centre of most computing devices and needs high amounts of energy. A common technique to reduce the processor's energy consumption is to change its computation speed depending on how much work the mobile phone currently needs to do. The slower a processor works, the less energy is needed to drive it. However, when the speed is too low, mobile users can perceive applications as slow and unresponsive. When the speed is higher than necessary, energy is wasted. An efficient processor speed selection technique must be able to find a good balance between those two extremes to save energy and satisfy the end user.

This thesis presents methods and tools necessary to develop such a technique. In the first part of this thesis, a method is introduced which can automatically decide if a particular processor speed selection technique does a good job in terms of user satisfaction and battery consumption. It is used to find the theoretical maximum amount of possible processor energy savings which can be achieved without slowing application responsiveness to a degree noticeable by the mobile phone user. In the second part a processor speed selection technique is developed which directly considers how a user perceives application responsiveness. It automatically learns which speed settings lead to a good balance between application responsiveness and energy consumption. In so doing, it manages to improve processor energy efficiency compared to current standard techniques whilst providing a satisfying experience to the end user.

Acknowledgements

Bringing this thesis to an end was most likely one of the most challenging tasks I have done so far in my life. Not only did I learn much about the topic I was studying over the last years, I also learned a lot about what it means to be a computer scientist in general. More importantly, writing this work taught me dedication, endurance, dealing with ups and downs, managing a seemingly endless workload and balancing my life between work and my family. (My English is probably still not perfect but most likely improved a little.) I am most grateful to everyone who accompanied me during my time as a postgraduate student and helped me with advice, feedback and simply friendship.

I would like to thank my supervisors Dr Hugh Leather and Dr Björn Franke for teaching me what it means to do actual research. Most of all for constantly pushing me towards becoming more independent and more confident with my own work. I consider myself very lucky for having met all those amazing people in and around the Pasta office 1.34 who were working with me and enduring my endless questions. This list includes but is certainly not limited to: Tom Spink (a big THANK YOU for prove reading my thesis!), Harry Wagstaff, Stephen Kyle, Igor Böhm, Kuba Kaszyk, Christopher Thompson, Matthew Bielby, Oscar Almer, Tobias Edler von Koch, Bruno Bodin and Pavlos Petoumenos.

I would like to express my gratitude to all the wonderful people from *Arm* for their generosity towards funding my postgraduate studies. Especially, I would like to thank my *Arm* mentors from the Processor Power Group Achin Gupta and Robin Randhawa for their support over the course of my studies and during my time in Cambridge. I also want to thank Dietmar Eggemann for being the best *Arm* buddy I could hope for and for putting up with my constant questions. A very special thank you goes to Dr Zheng Wang from the University of Lancaster for helping me to finish my thesis with additional funding.

Last but most certainly not least, I will like to thank my family, especially my wife, fellow scientist and best friend Luise Seeker. Without your constant support and open ear for my joys and sorrows, I would not be were I am today.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material in this thesis has been published in the following paper:

- **V. Seeker**, P. Petoumenos, H. Leather, and B. Franke. "Measuring QoE of Interactive Workloads and Characterising Frequency Governors on Mobile Devices.", in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization* (IISWC'14), 61-70, 2014. doi:10.1109/IISWC.2014.6983040.

(*Volker Seeker*)

To my kids Connor and Maya.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the mobile market growing stronger in recent years mobile devices have become ubiquitous in our daily lives. According to *Gartner* [1], 344 million smartphones were sold to end users globally in the second quarter of 2016, which is a 4.3% increase over the same period in 2015. That growing presence of mobile devices leads to a growing application of those systems to handle every day tasks both in personal and professional environments. With more involvement of mobile computing in everyday tasks, industry answered this trend by developing devices with more compute power. Unfortunately, battery technology does not scale with the same speed as computational power. Hence, the battery life of modern phones is drastically shorter compared to mobile devices from 10 years ago. When a Nokia 3210 developed in 1999 had enough power for a week, modern smart phones such as an iPhone 6 or the Samsung Galaxy S6 often last for less then a day before their batteries need to be recharged.

What is the culprit of this high energy demand? Next to powerful cellular and wireless networking modules, large and bright screens and complex graphic accelerators, the central processing unit (CPU) is still one of the most power hungry parts [2–4]. Figure 1.1 displays the power consumption breakdown from a study by *Torchiano et al.* [4]. They measured the power consumption of different mobile devices for various usage scenarios. Each scenario was designed in a way that it would make extensive use of a particular device component. Among those components were cellular and wireless networking modules responsible for phone calls and internet connection, namely 3G, 2G and Wifi, as well as CPU, audio or the display. According to their data, collected for a Galaxy Nexus S, 3G and 2G modules have the highest power consump-

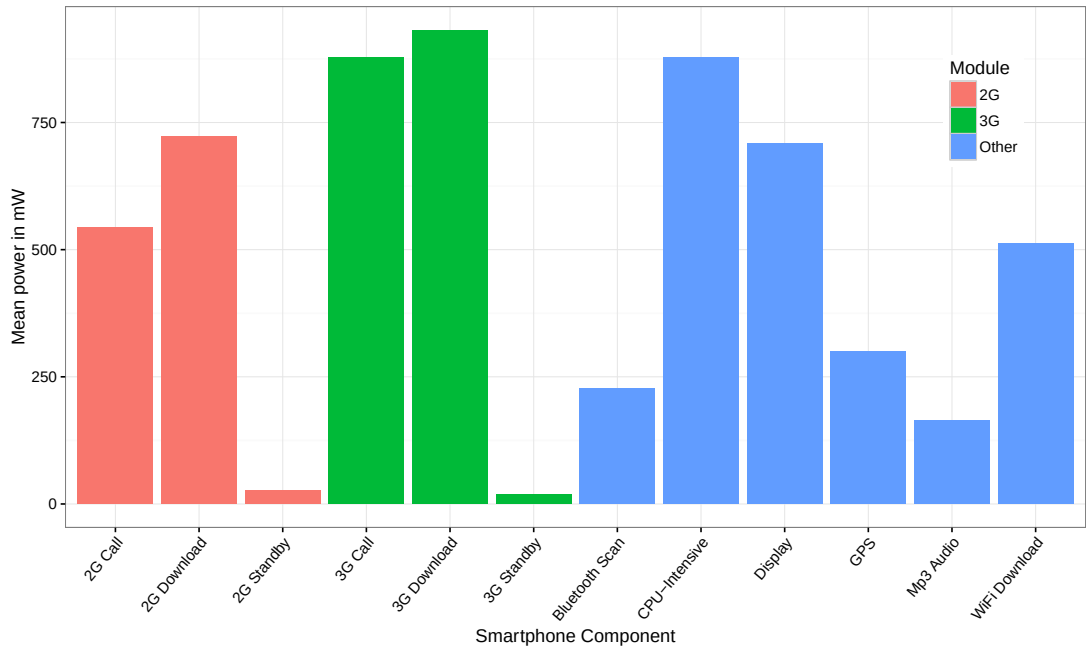


Figure 1.1: Galaxy Nexus S power consumption breakdown for different usage scenarios. Each bar represents a usage scenario which makes intensive use of a specific module.

tion, closely followed by the CPU and the display. With the goal of prolonging battery life time, this thesis will focus on improving the CPU's energy efficiency.

The two most prominent techniques developed to reduce the CPU's energy consumption are on one hand using various sleep states while the device is idling and on the other dynamic voltage and frequency scaling (DVFS) when it is busy. DVFS will be in the centre of this thesis.

1.1 Dynamic Voltage and Frequency Scaling

DVFS is a widely used technique to save CPU energy and prolong battery life. A DVFS algorithm lowers energy consumption by dynamically reducing the CPU's supply voltage needed to power its logic elements. However, reducing voltage means increasing circuit delay and the processor logic cannot keep operating at the same clock frequency. The clock frequency of a CPU determines how many instructions per time unit can be processed. Hence, reducing voltage also requires a reduction of the CPU's operating speed. A slower CPU can result in system performance reduction. To mitigate any negative performance impact the initial idea behind DVFS was to exploit

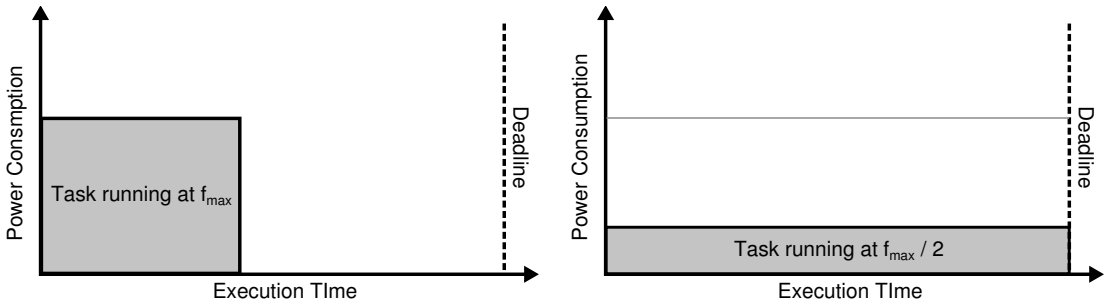


Figure 1.2: The same task is executed on two different voltage and frequency levels. The left side shows the task being executed on the maximum frequency. It finishes well before its deadline. The right side shows the task's execution with half that frequency. Here, execution ends right at the deadline. Due to a quadratic relation between voltage and dynamic core power, energy is significantly reduced (source [5]).

execution slack of deadline driven tasks. It is depicted in Figure 1.2.

In Figure 1.2 the same task is executed for two different voltage and frequency levels. The task needs to be finished by a given deadline which is indicated by a dashed line. On the left side the task is executed on the highest CPU frequency and finishes well before it is due. The corresponding power consumption is depicted on the y-axis. The area of the shaded box represents the energy consumption needed for the task execution. On the right side the execution frequency is only half of the maximum. The task needs longer to finish but still finishes before its deadline. This execution profile uses the CPU more efficiently over the available time by leaving no idle period. More importantly, dynamic core power scales quadratically with supply voltage (see Section 2.2 for details). This means even small voltage reductions lead to significant energy savings. Therefore, the right hand side scenario in Figure 1.2 leads to a drastic energy reduction.

Applying the approach demonstrated for a simple example in Figure 1.2 for a realistic mobile workload is a complex and challenging task. In a realistic mobile workload a multitude of different tasks with different execution statistics are running at the same time. They influence each other's runtime by competing for execution resources. Different tasks can have different runtime characteristics: They can be CPU intensive by executing instructions on the CPU most of the time, they can be memory intensive which means they often stall while data is being moved between memory and CPU, or they show a mixture of both properties. Changing the CPU frequency under those conditions leads to performance changes which are hard to predict. Current standard DVFS techniques on mobile systems tackle this problem using a CPU load driven heuristic.

On a *Linux* based mobile operating system (OS) such as *Android* the responsible module for DVFS is called a CPU frequency governor. Current standard governors on mobile devices make frequency decisions based on how much the CPU has to work at the moment, i.e. the ratio between processor task execution time and processor idle time. If this ratio raises above a threshold the execution frequency is raised. When the CPU load drops the frequency is lowered again. The *Linux* frequency governor was developed to work well for a broad range of different workloads, on servers, desktops or mobile devices. This thesis, however, is driven by the assumption that in the case of an interactive mobile workload a mainly CPU load based DVFS heuristic does not achieve the best possible energy efficiency. In Chapter 5 an *Oracle* study shows that for an interactive workload, standard mobile frequency governors need up to 32% more energy than an *Oracle* driven frequency selection. This study claims, that the user interaction intensive nature of mobile workloads allows significant room for energy efficiency improvements using DVFS. A user perception driven frequency governor developed in Chapter 7 is able match the *Oracle* profile's energy consumption much closer than standard governors and lies only 9.6% above it. The following section will point out where this energy saving potential lies.

1.2 Quality of Experience

The ultimate optimisation goal when tuning a mobile system is to provide a high quality of experience (QOE) to the end user.

Quality of Experience *In the literature QOE is used as a metric to measure the total system performance based on the end user's satisfaction.*

Satisfaction can be measured by observing the user's stress level [6], polling opinion based questionnaires [7], measuring the stability of video playback frame rate [8] or the network connection speed [9]. All of those metrics to measure QOE ultimately consider observations of the executed workload taken from the *user's point of view*.

Figure 1.3: User perception channels of mobile device output. Visual, audible and haptic.

The user's point of view *The user's point of view or the user's perspective shall be defined in this work as the system output as it is perceived by the user. A mobile system uses output devices such as screen, speakers, vibration modules or LED light indicators as an interface to the user (see Figure 1.3).*

Changing system parameters such as processor speed, memory bandwidth or cache sizes can have a significant impact on instruction throughput and task execution times. However, these changes are not necessarily perceptible from the user's point of view because they might not affect system output that is important to the user. This thesis claims that in the user's perceptibility of performance changes lies the key to better DVFS. When improving DVFS with the goal of maintaining a high QOE, it is important to know which perceptible system output is of significant importance to the user and which output is not. DVFS can then be optimised using different strategies for system output with high influence on QOE and output with low influence.

For interactive workloads the human computer interaction (HCI) research community identified the time a system needs to respond to a user interaction as of significant importance to the end user.

System Response Time *System response time (SRT) is the time a system needs to respond to a user interaction. It starts with user input and ends when the user feels that the system has finished servicing his*[1] *request. The remainder of this thesis will use the term SRT or interaction lag interchangeably.*

A large body of research work covers the search for a good SRT to satisfy the end user (see Section 2.3). A good value depends on various factors such as the frequency or complexity of the task initiated by the interaction or if system feedback is provided while waiting for the final result. Current standard DVFS techniques work well for mobile workloads when it comes to providing fast system response times. The energy saving potential lies in knowing how fast is fast enough to keep user perceived performance on an acceptable level.

Research Question *In a study by Dabrowski et al. [10] the author's state that computer systems these days are fast enough to handle most common tasks. With battery powered devices a new question emerges: How much can you slow a system response down with the goal of saving energy to maintain a good QOE?*

[1]The author of this thesis is aware that there can be male and female users. For simplicity, if a user is addressed in the remainder of this document he will be considered to be male.

Research Claim *A DVFS technique for interactive mobile workloads can accomplish good energy efficiency if information is available on how fast the system responds to interactions as seen from the user's point of view. Such kinds of information helps to evaluate the effect of frequency changes on user perceptible performance and allows to find frequency settings fast enough to maintain acceptable QOE whilst still achieving energy savings.*

1.3 Motivating Example

The claim driving the research of this thesis is that DVFS for an interactive mobile workload has the potential of gaining higher energy savings if the DVFS governor is considering the user's point of view, i.e. the workload execution as a user would perceive it. Data on when device output is important to end users to achieve high QOE helps to make energy efficient frequency decisions. This section supports this claim by pointing out potential energy savings for an example of two specific interactions with the device. The example is taken from the benchmark workload generated later in this thesis (see Chapter 4). In the example, frequency choices of the *Ondemand* frequency governor are compared to an *Oracle* frequency profile. *Ondemand* is the current standard governor on many mobile devices. A technique to generate *Oracle* frequency profiles for interactive mobile workloads will be presented in Chapter 5.

Figure 1.4 shows the execution of the two interactions including relevant system statistics. The executed content shows the *Pulse* news application. Relevant screenshots are shown in the centre of the figure. A finger indicates where an interaction happened by touching the device's screen. The first screenshot shows an article selection of the news application. The screen is touched at the first article on the top left which opens the corresponding text. After a transition animation between menu and article text, the second screenshot is visible and the corresponding content can be read. From the user's point of view, this indicates the ending of the first interaction lag and the user feels that the system has finished processing the interaction. After he finished reading, he presses the back button on the bottom of the screen. This is the second interaction. It results in the application going back to the article overview and ends when the overview is fully visible again.

The bottom of Figure 1.4 shows the frequency profiles selected by *Ondemand* (thin

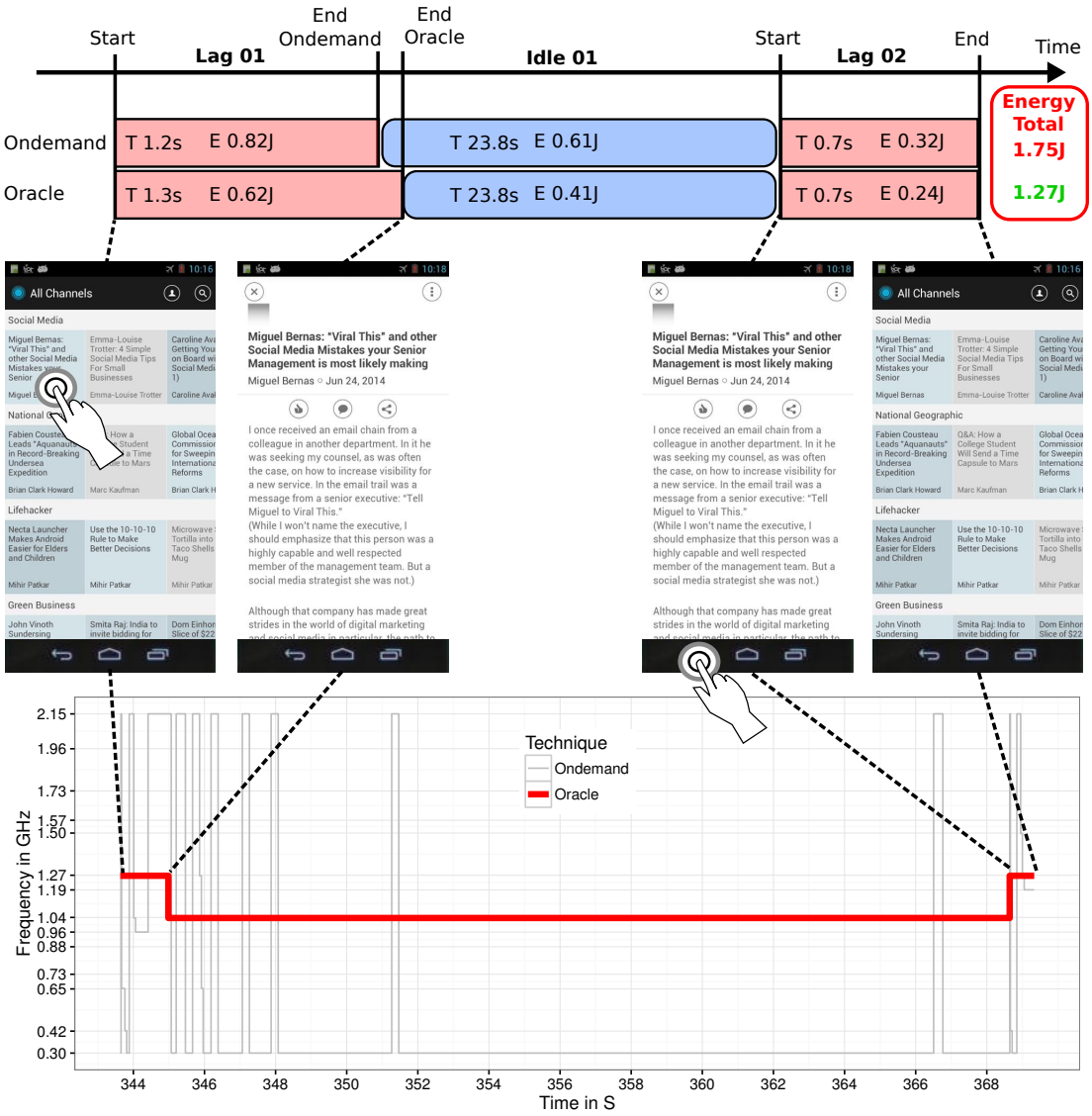


Figure 1.4: Comparison of *Ondemand* and *Oracle* frequency governor performance profiles for an interaction example. The screenshots in the centre show two consecutive interactions: Selecting a news article from an overview and pressing the back button while the article is displayed to return to the overview. The graph on the bottom shows the frequency selection of *Ondemand* and *Oracle* governor. The boxes and timeline on the top of the figure indicate duration and energy values for the different phases of the interaction executions for both governors.

grey line) and the *Oracle* governor (thick red line). *Ondemand* raises the frequency soon after the first input arrives and alternates between highest and lowest frequency while working on opening the article. During the reading phase a few frequency spikes appear around the end of the first interaction and towards the beginning of the second one. When the second interaction occurs, *Ondemand* raises the frequency again to process it. *Ondemand* selects frequencies from the whole available spectrum but mostly

focuses on the highest and the lowest frequency. The *Oracle* governor only uses two frequency levels. 1.27 GHz during the interaction lags and 1.04 GHz during the reading phase. The specifics of why these frequencies were selected will be explained in Chapter 5, where the *Oracle* governor is introduced.

The top of the figure shows a timeline marking start and end of the two interaction lags. The end is marked as seen by a user, i.e. when he feels that the system has completely processed his input. This does not necessarily correspond with the CPU being busy or not. The red boxes with sharp corners indicate the duration of the lags while the blue boxes with rounded corners show the duration of the reading period, i.e. where the system appears to be idle. Inside the boxes a T marks the corresponding duration and an E marks the energy consumption. The box on the top right shows total CPU energy consumption of the *Ondemand* governor's frequency choices compared to the *Oracle* ones. Even though the difference in execution durations is minimal, the *Ondemand* governor uses 27% more energy.

This example demonstrates for two scenarios where the *Ondemand* governor fails to exploit energy saving potential and the *Oracle* is more energy efficient:

Overshooting The Goal A first potential energy saving can be seen when looking at interaction lag times. The execution durations only differ for the first interaction. Here the lower frequency selection of the *Oracle* compared to the high spikes of the *Ondemand* governor leads to an increase of SRT by 100ms. According to various HCI research sources [11–13], this difference is clearly below the perceptible limit of a user, i.e. he is unable to notice this difference. The second interaction even has the same duration for *Oracle* and *Ondemand*. Still the *Oracle* needs less energy by choosing a lower frequency. This indicates that *Ondemand* is overshooting the performance settings for user interactions and could achieve better energy efficiency without noticeable effect on QOE if selecting lower frequencies.

Imperceptible To Users A second energy saving potential can be found in the idle period. Even though the user already concluded that the interaction is over and started reading the text, *Ondemand* is raising the frequency repeatedly. This is due to background tasks of the application or the OS which require CPU computation time. *Ondemand*'s heuristic raises the frequency based on CPU load and does not consider if the user is actually able to perceive whether the current

load is executed quickly or slowly. In the example the presented text is static, no other output devices are being used. The medium frequency choice of the *Oracle* is perfectly capable of handling all background tasks without affecting the user's QOE in a negative way[2]. In so doing it uses 33% less energy during the idle period. This indicates that *Ondemand* is selecting too high frequencies at times where a difference in CPU performance is imperceptible to the user.

This example taken from the benchmark workloads generated in Chapter 4 shows how current standard DVFS techniques on mobile devices fail to exploit energy saving potential in two cases. An *Oracle* approach having full knowledge of the user's point of view can tap this potential. It does so by supplying the right amount of performance to maintain acceptable QOE during interaction lags and keeping the performance level low during idle periods where performance differences are imperceptible to the user. The experiments presented in this thesis will demonstrate how energy saving potential of this kind can be identified systematically for a representative mobile workload and how it can be tapped at runtime to outperform the energy efficiency of current standard DVFS techniques on mobile devices.

1.4 Contributions

In pursuit of extending mobile devices' battery life with a more energy efficient and user perception aware DVFS technique, this thesis makes the following four contributions:

QOE Benchmark A methodology is presented to benchmark representative interactive mobile workloads in terms of interaction lag dependent QOE.

Energy Efficiency Oracle Study Using the QOE benchmark an *Oracle* study is conducted which identifies energy saving potential in an interactive mobile workload. The *Oracle*'s results are compared to current standard DVFS governors.

Runtime Interaction Lag Detection Heuristic A runtime heuristic is developed to capture start and end of system responses as seen from the user's point of view. This heuristic lays the foundation for an improved DVFS algorithm.

[2]One could ask why the *Oracle* does not pick the lowest frequency if the user does not notice an effect. As can also be seen later in Chapter 5 and as mentioned in Section 2.2.2, the lowest frequency is not necessarily the least energy consuming one if the CPU is busy. Sprinting towards idle can need less energy than executing with the lowest frequency and voltage.

QOE Driven DVFS Algorithm Together with the lag detection heuristic, a DVFS algorithm is presented which uses a reinforcement learning technique to tap the energy saving potential identified in the *Oracle* study whilst maintaining good QOE. It is evaluated using the QOE Benchmark.

1.5 Thesis Roadmap

Chapter 2 will provide more information on DVFS and will cover the most prominent research areas on that topic. Furthermore, this chapter will describe the concept of SRT as seen by the HCI research community and will summarise research development in that area. This is followed by a description of the reinforcement learning method. Additionally, this chapter will cover the Android Open Source Project with focus on the graphics framework and lastly current standard CPU frequency governors of mobile systems will be introduced.

Chapter 3 will cover closely related research on benchmarking for interactive mobile workloads, specifically considering SRT. Also, related user perception and machine learning driven DVFS studies are discussed.

Chapter 4 describes a methodology which is able to record and replay representable interactive mobile workloads. Furthermore, a novel technique is introduced which considers the device's screen output to measure interaction lag of workload executions as seen by the user. Comparing lag measurements of the same workload executed for different system configurations allows benchmarking the system in terms of QOE. This benchmark is used in the following chapters to evaluate DVFS techniques.

Chapter 5 uses the QOE benchmarking methodology from Chapter 4 to conduct an *Oracle* study. It generates frequency profiles for a benchmark workload with maximum energy savings and minimal user irritation. A novel user irritation metric is presented which can give an irritation score to a workload execution by analysing measured interaction lag durations. The *Oracle*'s frequency profile will be used as a baseline for DVFS optimisations performed later in the thesis. Additionally, the *Oracle* results are used to evaluated how well current *Android* frequency governors are performing.

Chapter 6 presents a detailed study of system characteristics during interaction lag periods as seen from the user's perspective. The collected insight is used to devise a

heuristic which is able to track the start and the user perceived end of interaction lags in a mobile workload at runtime.

Chapter 7 uses the interaction lag detection heuristic from the previous chapter to develop a QOE aware DVFS governor. The algorithm is based on a reinforcement learning method. It uses a trial-and-error approach to learn optimal frequency settings for encountered interaction lags in an interactive mobile workload. The presented governor is evaluated using the benchmark and irritation metric from Chapter 4 and Chapter 5, and compared to the results of the *Oracle* study and standard governor evaluation.

Chapter 8 will summarise the results presented in this thesis, draw a conclusion and discuss future work.

Chapter 2

Background

2.1 Introduction

This chapter will present background knowledge on the concepts and techniques developed and studied in this thesis. It is separated into five parts:

Dynamic Voltage and Frequency Scaling (DVFS) The research goal of this thesis is to improve DVFS energy efficiency for an interactive mobile workload to prolong battery life. DVFS is a technique to dynamically control processor clock speed and voltage to reduce CPU energy consumption. Its concepts and research background will be described in part one.

System Response Time (SRT) End user perceived time intervals needed by the system to respond to his input, are considered in this thesis to identify critical execution phases. The second part of this chapter will give information on research in the field of user response time in interactive workloads. It will focus on how response time length affects end user QOE.

Reinforcement Learning (RL) The following section will describe the concept of reinforcement learning. This machine learning technique learns optimal system behaviour with a trial-and-error approach. It is used in Chapter 7 to create a user perception aware frequency governor by learning how different frequencies affect SRT durations for different user interactions.

Android Open Source Project The Android Open Source Project (AOSP) provides operating system sources to developers for analysis and modification. Experi-

ments conducted in this study rely on instrumentations made to the *Android*'s graphics framework. For a better understanding, background information on AOSP will be presented.

CPU Frequency Governors The DVFS strategy developed in this thesis is compared against current standard implementations on mobile devices. Details on current standard CPU frequency governors will therefore be presented in the last section of this chapter.

2.1.1 Terminology

To avoid confusion about the usage of the terms energy consumption and power consumption in the remainder of this thesis, they shall be defined here:

Energy Consumption Energy or more specifically electrical energy is consumed over time. It is usually measured in joules (J) or kilowatt-hours (kWh).

Power Consumption Power consumption is defined as the amount of energy consumed in an instance of time. It is measured in Watts (W).

2.2 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a technique to reduce the CPU's energy consumption while the processor is busy. The general approach is to dynamically reduce CPU processing speed and energy consumption in non-critical execution phases. The two main challenges are on one hand to identify critical and non-critical phases in a given workload and on the other to predict how frequency changes affect the system's performance while executing that workload. The following section will give information on the micro-architectural background for the power breakdown of a modern CPU. It will further describe which power saving techniques are commonly applied and where DVFS can be of use.

2.2.1 CPU Power Dissipation

The power dissipation of a computer system's central processing unit (CPU) can be subdivided into three parts: dynamic power consumption (P_{dyn}), short circuit power consumption (P_{short}) and leakage power consumption (P_{leak}) [5, 14, 15]. The following power equations are often found in literature on integrated circuits using complementary metal-oxide-semiconductor (CMOS) technology. They describe these three components. Total power is defined as:

$$P = P_{dyn} + P_{short} + P_{leak} \tag{2.1}$$

Dynamic Dynamic power consumption occurs during capacitance charging and discharging of CMOS logic circuits. It is defined as:

$$P_{dyn} = ACV^2 f \tag{2.2}$$

C is the total load capacitance, A is the average number of logic switches per clock cycle, V is the supply voltage and f the frequency of the clock.

Short Circuit Short circuit power is expended when a short circuit current I_{short} flows for a short time τ during logic gate switches. It is defined as:

$$\tau AV I_{short} \tag{2.3}$$

Leakage Leakage power dissipates because a small leakage or subthreshold current I_{leak} flows through transistor layers even when CMOS transistors are turned off. It is defined as:

$$P_{leak} = V I_{leak} \tag{2.4}$$

Over the years multiple power saving strategies have been developed to make processors more energy efficient, specifically for embedded systems and server farms. These strategies focus mostly on four areas:

DVFS DVFS allows to dynamically vary operating frequency and supply voltage of logic elements to reduce dynamic power dissipation.

Dynamic Power Management This technique is closely coupled with DVFS and enables the CPU to enter different low-power states during idle time.

Thermal Management Here, the focus lies on the thermal aspects of the processor. Thermal management aims to reduce heat dissipation to save power used by cooling elements.

Heterogeneity Aware Scheduling This approach is considered when a processor comprises multiple cores of different complexity and thereby different power levels. Intelligent task scheduling is able to save energy by running tasks with low computational intensity or priority on small and energy efficient cores while tasks with high computational intensity or priority are executed on high performance ones.

In this thesis, a technique for mobile devices is developed to save energy by dynamically adjusting the frequency and voltage level. Therefore, the following sections will focus on functionality and background literature of DVFS techniques.

2.2.2 DVFS Approach

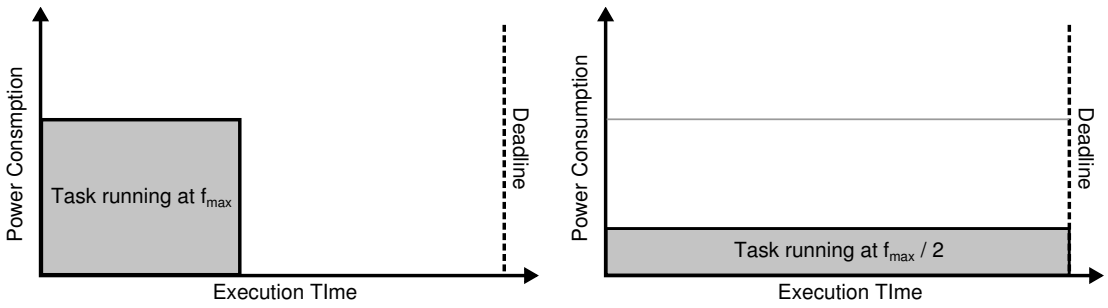


Figure 2.1: The same task is executed on two different voltage and frequency levels. The left side shows the task being executed on the maximum frequency. It finishes well before its deadline. The right side shows the task's execution with half that frequency. Here, execution ends right at the deadline. Due to a quadratic relation between voltage and dynamic core power, energy is significantly reduced (source [5]).

DVFS is an energy management technique to dynamically reduce the supply voltage of CMOS logic gates in a CPU to save energy. According to Equation 2.2, reducing voltage leads to a quadratic reduction in power consumption. This means, even small voltage reductions can bring significant energy savings. However, since reducing voltage also means increasing circuit delay, the processor logic cannot keep operating on the same clock frequency. Therefore, reducing voltage also requires a reduction in operating frequency which has an effect on system performance.

The key idea behind DVFS is shown in Figure 2.1. Here, DVFS is used based on the observation that finishing a task early and then idling until its deadline, is less energy efficient than finishing the task as close as possible to its deadline. On the left side of the figure a task is executed with maximum core frequency and finishes well before the deadline, indicated by the dashed line. On the right side the frequency is reduced by half which reduces execution speed and allows lower voltage. Now, the task hits the deadline precisely. Following this naive picture, the energy consumption is quartered for the execution on the right side due to the quadratic relation between dynamic power and voltage. Still, the corresponding task finishes in time, despite its prolonged execution duration.

This demonstration of a DVFS application is highly simplified. The actual effect of DVFS on a system's performance and energy consumption under its current task load is usually not as easily determined. Many real-world complexities need to be considered [15]. A big problem is the unpredictable nature of workloads. It is hard to estimate which tasks will appear at any point in time and what their execution time and characteristics will look like. Furthermore, real-world systems show a number of non-determinism and anomalies in the relation between clock speed, task execution time and energy consumption:

System Power The power dissipation of a microprocessor might be quadratic to its supply voltage but that is usually not true for the power of the entire system. A task might use an energy demanding component such as a cellular module. If its execution speed is reduced, the CPU might use less energy, but keeping the component active for longer might lead to a higher energy consumption of the whole system.

Execution Speed There is a long debate between researchers whether it is actually more energy efficient to finish a task close to its deadline, aka. *pacing*, or if more energy can be saved by finishing the task as quickly as possible to enter a deep sleep state early, aka. *sprinting*.

Context Switches Most modern systems do not run a single task on a single CPU but rather constantly switch contexts on multiple cores between multiple tasks. Changing execution speed can lead to rearrangements in future context switches which are hard to predict.

These and other complexities have led to a large body of research work where scien-

tists looked into various ways of optimising DVFS application for real-world systems. Based on [5, 15, 16], the following sections will give an overview of past research work. Studies from areas closely related to the techniques applied in this thesis, such as user perception or machine learning driven DVFS, will be discussed in more detail in Chapter 3.

2.2.3 DVFS Research

The earliest work on DVFS was done by *Weiser et al.* [17] in 1994 and a year later extended by *Govil et al.* [18]. In this early work the authors conducted experiments testing various scheduling techniques which would dynamically reduce operating frequency and voltage to slow down tasks so they would hit their deadline more closely. They reported 50-70% energy savings after evaluating their scheduling techniques using trace-driven simulation.

In a survey from 2005, *Venkatachalam et al.* [15] subdivided DVFS algorithms into three major categories:

Interval In the interval based approach, the time a processor is busy is measured over the timespan of a given interval. Based on observed statistics, future utilisation and required frequency settings are determined. Examples for interval based DVFS can be found in [17–20]. Most current standard DVFS implementations for *Linux* based systems rely on the interval approach (see Section 2.6).

Inter-Task The inter-task approach assigns processor speed settings on a task granularity as for example in [21, 22]. For each task a frequency setting is chosen based on previous observations of this task's execution characteristics. At each context switch to the task, the CPU is fixed to the selected frequency.

Intra-Task The intra-task approach increases granularity by additionally considering phase changes within single tasks. Researchers subdivided single tasks in subregions based on program structure or measured execution characteristics. They assigned DVFS levels to each identified region. Examples can be found in [23–27]. Other policies to specify subregions [28–30] were implemented on a compiler level. They considered program execution paths or checkpointing.

Trying to exploit the deadline approach depicted earlier in Figure 2.1, many studies are focusing on real-time systems. Here, task deadlines were usually known beforehand

in the form of a worst case execution time (WCET) [31–37].

Many introduced DVFS algorithms make use of a workload's micro-architectural characteristics to estimate performance loss when reducing execution speed [38–46]. One significant indicator for low performance losses is the memory-boundedness of a task. If a task has long stall times while waiting for memory accesses, CPU frequency can be reduced without significant impact on the task's execution speed. In contrast stands a task's CPU-boundedness for which researchers showed a linear dependence between processor frequency and execution time. A good balance needs to be found if a task shows a combination of both characteristics. One of the major challenges in this area is to detect at runtime whether a task is memory- or CPU-bound.

In contrast to a global application of DVFS where the clock speed of the entire processor is manipulated, methods were evaluated to set voltage and frequency on a more fine grained level. The practice of fine grained DVFS application was first analysed with the introduction of multiple clock domain processors in 2002 [47]. Single components of the processor such as floating point unit, integer unit, memory, etc. were driven by their own clock and voltage supply, i.e. had their own *voltage-frequency islands*. This architecture design made it possible that only components under heavy load were running on higher frequencies using more energy. Many studies were conducted on refining DVFS techniques for processors with multiple clock domains. This was initially done for single core processors [48–53] and later extended to machines with multiple cores [20, 54–56].

In this thesis experiments are conducted for single core configurations to reduce statistical error due to load balancing. For the sake of a comprehensive picture, however, DVFS research background on multi-core systems will still be covered here briefly. Multi-Core processors or chip multiprocessors (CMP) brought a new set of challenges for energy efficient DVFS applications. Especially, when frequency levels can be set separately for each core or groups of cores. By simultaneously setting different cores to different frequencies, heterogeneity is artificially introduced into the system. Further heterogeneity among actually homogeneous cores comes due to chip manufacturing variations. Now not only energy efficient DVFS settings, but also good thread to core mappings need to be found. DVFS research on implicit heterogeneity and power efficient thread to core mapping was done by [57–61].

With CMP systems also came the problem of shared resource contention. Multiple

cores of a single processor often share resources such as last level cache, prefetching hardware, memory bus, etc. Competing for shared resources can lead to significant slow downs of stalling tasks. In that context, researchers analysed how resource contention can be reduced and energy efficient scheduling and DVFS applied at the same time [62–65].

As mentioned in Section 2.2.2, DVFS can only be used to reduce dynamic power consumption. With shrinking feature size of modern processors their static component, i.e. leakage power dissipation, grows. Observing this trend as well as how the improvement of processor's deep sleep modes, researchers argue that potential energy savings of DVFS will hit a limit or even diminish in the future [66–69].

2.2.4 Summary

DVFS is a technique to reduce the power dissipation of dynamic core power. This is done by carefully adjusting supply voltage and clock frequency depending on execution demands of the current workload. The goal of this technique is to reduce energy consumption without major performance impact. Researchers analysed various approaches, changing frequency settings on interval, task or program phase granularity. To find the right performance level, task characteristics were being observed using offline profiling or online sampling methods. The multi-core era brought additional complexity by adding the dimension of finding good DVFS settings for multiple cores and multiple active program threads at the same time. With shrinking feature size of modern processors researchers predict a limit to DVFS energy savings due to growing leakage power.

The DVFS technique presented in this thesis makes frequency decisions on a workload execution phase granularity. Execution phases are indicated by system response intervals to user input as seen by the user. The presented approach finds the correct frequency level by using a reinforcement learning mechanic to evaluate observations of multiple response time samples. Related user perception based DVFS techniques driven by machine learning are discussed in Section 3.3.

2.3 System Response Time Effects on QOE

The DVFS approach taken in this thesis relies on information about when performance changes are actually perceptible by the end user and when they are not. Human computer interaction (HCI) research identified the time it takes the system to respond to user input as of significant importance to maintain high QOE. Therefore, the effect of frequency changes on system response time (SRT) as seen from the user's point of view are considered in this thesis to drive DVFS optimisations. The following section will give an overview on SRT research background to provide a context.

2.3.1 System Response Time Concept

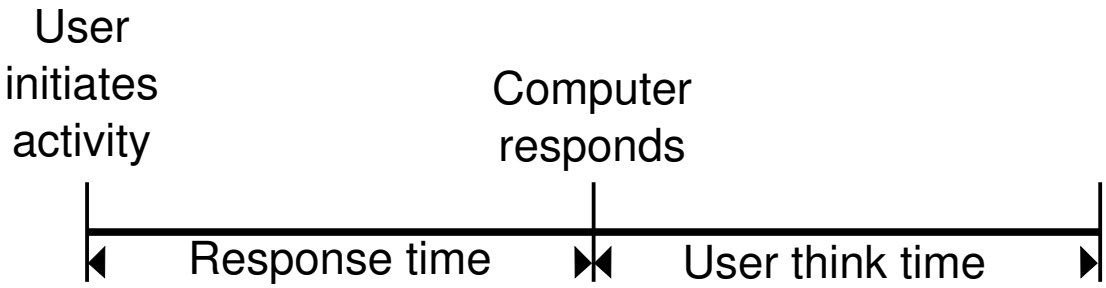


Figure 2.2: Simple model of user interaction, system response and user observation of presented results [12].

User interactions with computer systems usually follow a specific pattern. This pattern is depicted in Figure 2.2 in the form of a simple model developed by *Shneiderman* in 1984 [12]. First, the user initiates an action by providing some kind of input to the system, e.g. key press, mouse move, voice command, etc. The computer system processes the input and responds to the given command. Depending on the initiated action the user might have to wait some time until results are being presented. As soon as the system starts displaying a response the user can observe it and start thinking about what to do next. He might then decide to initiate another action and the cycle starts all over.

The amount of time between initiated action by the user until the system finishes displaying results on the screen or printer is called system response time (SRT). In some research work SRT only refers to the time until the system shows a first input acknowledgement even though it did not yet finish displaying the entire response. In Figure 2.2, the time period following a response is labelled user-think-time and indicates the times-

pan where the user thinks about what to do next. In reality user-think-time might look more complex. Users are often planning ahead several steps while initiating an action, they might not care about the output or cancel it altogether. In [12] Shneiderman describes a more realistic model of user-think-time and discusses it in detail.

SRT is at the centre of research work since the 1960s. Scientists are evaluating various parameters to determine which response duration in what context leads to optimal QOE for the user. In the earliest known work on SRT, *Miller* [70] hypothesised that user interactions with a system can be seen as a conversation between two persons. As long as certain limits in response time are not violated and no inconvenient pauses occur the conversation keeps flowing. He identified 17 system interaction categories with different response time requirements. Those requirements are to be met if the user's work flow should not be disturbed.

2.3.2 Research Work on Different SRT Aspects

Later studies evaluated and extended Miller's early guidelines. In 2011 *Dabrowski* [10] presents a detailed review of SRT research over the course of the last 40 years. The following paragraph will give a short summary of its most important aspects:

Errors Several studies [6, 71–81] describe experiments to determine if varying SRT has an effect on the mistakes a user makes while working with a computer system. Mistakes such as entering wrong data or clicking a wrong user interface (UI) element. It turns out that even a small SRT delay on an atomic level, like during key presses or mouse movements, leads to immediate mistakes by users. If the system is not able to follow a user navigation, error rate goes up. If long delays happen only between tasks where delays are to be expected, less mistakes were being observed.

Productivity The completion rate of tasks or amount of work done over a period of time was analysed while modifying SRT. Many authors [6, 71–73, 75–77, 79, 80, 82–90] reported that in general, productivity decreases if SRT increases. Specifically, when the SRT affects interactions on an atomic level, i.e. for mouse movements or text input.

User Adjustment Researchers [71, 72, 74, 76, 78, 81, 85–87, 89, 91–93] analysed the ability of users to adjust their behaviour to changing SRT. The results show

that user-think-time (see Figure 2.2), aka. user-response-time, increased with increasing SRT. But also the type of commands users issued changed with longer SRT. In that context, scientists described SRT as a measure of task execution costs. The longer the SRT, the more time the user would spend on thinking about what to do next before investing a high cost again. He would also change the type of commands being issued. If the delays were shorter, users would often issue simple commands in a trial-and-error type of way. For longer delays they would use more complex commands to get more work done within a single SRT cycle.

Psychologial Effects Researchers [72, 73, 80, 86–88, 90, 94–103] looked into the connection between SRT and several psychological effects such as user irritation, boredom or the perceived quality of the system. Same as for productivity and error rate, user irritation increases and perceived quality decreases with growing SRT. Some of the negative effect can be mitigated by presenting the user with continuous feedback like animations or loading bars while the system is busy working on the response.

Physiological Effects Finally, a few studies [6, 80, 104–106] were conducted on the effects of SRT on physiological factors such as stress and anxiety. Scientists measured heart rate or blood pressure while varying inter task delay. The observed results showed that too long delays lead to heightened anxiety in users as well as too short ones. Following that, researchers argued that an optimum level of SRT can be found which gives the user an appropriate amount of time to prepare for the next task and allows him to finish it with low error rate and low anxiety.

The general conclusion which can be drawn from this body of work is that increasing SRT in most cases has a negative effect on the user. His irritation and stress level grows and he tends to make more mistakes, especially when delays happen on an atomic level. If long response times occur between tasks users are able to adapt up to a certain point. They tend to think longer about what to do next and issue more complex commands to make sure the time during the next delay is well spent.

2.3.3 SRT Taxonomies

Based on research findings over the years, scientists refined Miller's guidelines from 1968 and developed taxonomies of their own to specify good SRT boundaries depending on the context. The taxonomy being around the longest was introduced by *Shneiderman* in 1987 [12] (see Table 2.1a). It suggests four different levels of SRT depending on the complexity of the executed task. Atomic tasks such as typing or moving the mouse should have very short delays of 50 - 150 ms while more complex tasks were allowed up to 12 second delays before negative effects for the user would pick up.

Task	SRT
Typing, mouse movement	50 - 150 ms
Simple frequent tasks	1 s
Common tasks	2 - 4 s
Complex tasks	8 - 12 s

(a) Shneiderman's Categorisation

Expectation	SRT
Instantaneous	100 - 200 ms
Immediate	0.5 - 1 s
Continuous	2 - 5 s
Captive	7 - 10 s

(b) Seow's Categorisation

Table 2.1: Response time taxonomies after Shneiderman and Seow.

Another attempt to create an SRT categorisation was made by *Seow* in 2008 [13] and later confirmed by *Anderson et al.* [107]. Instead of considering task complexity, *Seow* categorised SRT levels by user expectations (see Table 2.1b). The first category, *Instantaneous*, refers to atomic events and allows with up to 200 ms slightly longer delays than Shneiderman's atomic event threshold. The *Immediate* category specifies optimal delays for interactions which require immediate acknowledgement by the system for some action executed by the user. The system might not be finished processing the initiated action yet but the delay between input and the first feedback should not be longer than 1 second. The third SRT boundary specifies a maximum delay in order to maintain a *Continuous* flow of user interactions, like in a conversation. Lastly, tasks started by the user where he simply has to wait for them to finish are covered with the last category, *Captive*. Here, Seow suggests some kind of feedback within 7 to 10 seconds if the user is to be kept engaged with the system and not abandon the task.

In 2015, *Doherty and Sorenson* [108] presented a third categorisation of SRT which combines both *Shneiderman's* and *Seow's* taxonomies. It adds the additional aspect of users' perceptional limits, such as attention span. Further approaches to quantify SRT were taken by *Tolia et al.* [109] who categorised how users of thin clients experienced response times due to network latency. More recently *Verheij* [110] introduced

quantification metrics for response time in a white paper in 2011. He considers task complexity and SRT variation in a desktop environment.

The presented research leads to the assumption that it is unlikely to find a universal SRT model or metric. Rather constant adaptations of traditional approaches are required for each new generation of computer systems and for different applications.

2.3.4 Perceived Performance

Once an acceptable SRT taxonomy is found for the problem at hand, the next step is to implement a way of applying it at runtime. Past research indicates that simply improving the performance of system components might not be enough to satisfy SRT thresholds and thereby the user. *Tim Mangan* was the first to introduce the term *perceived performance* in a white paper published in 2003 [111]. He refers to this term when analysing the system while applying performance improvements which the user can actually feel. These are standing in contrast to improvements which might increase the compute capacity of the system but are not noticeable by the user.

For a long time interface designers built their systems in a way that the first form of feedback following user inputs came no later than 100ms. In 2001 *Dabrowski et al.* [11] questioned this boundary and conducted experiments to test it. They found that in their experiments users did not notice delays of 150 to 200ms.

Making use of *Weber's Law* [112], *Seow* [13] suggested in 2008 a *20%-rule* when introducing performance differences which are meant to be noticed by the user. According to *Seow*, if a task increases or decreases its delay by less than this cut-off, the user will most likely not notice a difference.

2.3.5 Summary

The general trend for SRT goes towards shorter delays rather than longer ones. This trend reflects in developed taxonomies. Even though researchers introduced a variety of different models and metrics over the years, most interactions require a delay of a few hundred milliseconds up to a couple of seconds. Despite that, future systems often bring new ways of interacting with them. When 40 years ago most interactions were executed by pressing the RETURN key after entering a command [113], there are now

touch screen input, voice control or gyroscope movements. For each new context it is likely that the presented SRT taxonomies need to be adapted. More importantly, developers need to find new ways of implementing their software to keep interaction response times within suggested deadlines.

In Chapter 4 a methodology is developed to benchmark interactive mobile workloads considering SRT as seen from the user's point of view. Taxonomies and guidelines on perceived performance presented in HCI research are used in Chapter 5 to create a metric which helps to quantify QOE for said workloads. Closely related research on measuring SRT durations for mobile workloads will be presented in Section 3.2.3.

2.4 Reinforcement Learning

Chapter 7 uses a reinforcement learning (RL) driven technique to find good frequency levels for SRT instances. This approach allows the developed DVFS algorithm to adapt its frequency decisions to varying workloads. The following section will describe the general concept of RL and the specific approach applied in this thesis.

Reinforcement Learning is defined by characterising a specific learning problem. A learning agent learns which actions to take in a given situation to achieve a goal. This is not done by providing example training data, as for supervised learning. Rather, the agent has to learn optimal behaviour itself using a trial-and-error approach. *Sutton and Barto* [114] consider every algorithm that is able to solve a problem of given characteristic as an RL algorithm. The following description of RL is mostly according to their text book on the topic.

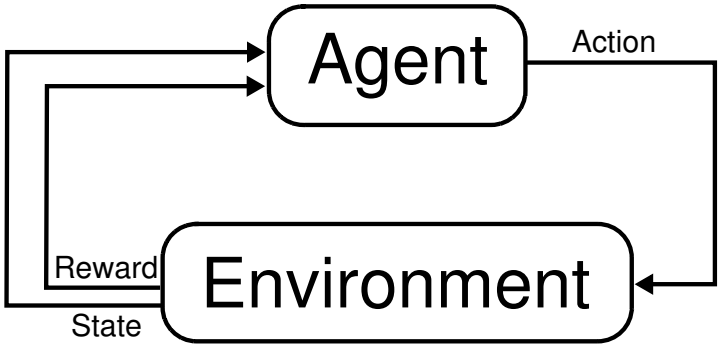


Figure 2.3: Interconnections between an RL agent and its environment. Each action executed by the agent affects the environment's state. Information on the current state and a reward value are processed by the agent to learn correct behaviour. (source [114]).

In RL systems an *agent* learns correct behaviour by interacting with its *environment*. Each *action* executed by the agent affects the environment and leads to a *state* transition. Information on the new state are passed to the agent as well as a *reward* value resulting from the last action. The reward indicates how well the executed action helped to achieve the desired goal. The agent evaluates the reward and adapts its behaviour if necessary. Depending on the environment's new state it decides which action to execute next. The interconnections between agent and environment are displayed in Figure 2.3.

Examples for RL can be found in many fields: On a very high level, RL is used in nature when a baby bird learns how to move its wings for flying or when humans practise hand-eye-coordination to pour milk in their cereals in the morning. A more technical problem needs to be solved for a building's heating system. An RL algorithm can use sensor data such as outside and inside temperature, weather forecast and room occupancy to learn correct regulations of radiator valves. RL is also successfully applied in robotics. There it is used to teach a robotic arm and hand how to grasp objects. It does so by learning the correct power levels of electric motors to move corresponding joints. It was also used by scientists for teaching robots how to play football [115]. To test out RL capabilities, scientists applied this technique to learn the rules of play for board or video games such as Backgammon, Go or Space Invaders. There, the implemented algorithms try to beat a human player or improve overall highscores [116, 117].

Next to agent and environment there are four additional main elements in an RL system:

Policy The agent's policy describes which actions are to be taken depending on a given state. This policy changes over time while the agent improves its behaviour.

Reward Function The reward function specifies the desired goal of the system. It maps pairs of executed actions and resulting states to a single number. This number represents how well the system was moving towards the desired goal with its last action. It reinforces the application of good behaviour. The agent's ultimate goal is to maximise the reward for each executed action.

Value Function While the reward function indicates good behaviour for the last executed action, the value function indicates good behaviour in the long run. It considers potential future states following the current one and is used for planning future actions over more than a single step forward. A greedy policy would

always select the action leading to an immediate optimal reward. This can lead to a local reward maximum but not necessarily to a global one. The value function allows to improve upon this behaviour by estimating multiple future steps where suboptimal actions in the immediate sense lead to higher rewards in the long run.

Environment Model In some learning systems a model of the executed environment is used for planning future actions. It mimics the environments behaviour for executed actions and is able to predict future states.

A major challenge for RL systems is to find a good balance between *exploration* and *exploitation*. The agent always faces the decision between executing an action which leads to optimal rewards according to its current estimations and executing an action to further improve its knowledge of the environment and thereby its policy. It is possible that the environment is stationary which would allow the agent to find a global optimum. If the environment changes its behaviour over time it is considered not stationary and constant improvement of the agent's policy is necessary.

This challenge is exactly what a DVFS technique faces that aims to reduce energy consumption while keeping QOE high. Both need to be balanced against each other. It is tackled by the work presented in Chapter 7.

2.4.1 Multi-Armed Bandit Problems

Optimising DVFS energy efficiency for independent user interactions falls into a specialised category of RL problems. If the agent needs to make a decision about which action to take next in only a single reoccurring situation, the original RL problem is simplified. In this setting distinct decision making situations are independent from each other. That means, assuming the policy is not changed, decisions taken in the past do not affect the decisions taken by the agent in the future. This is true for finding an energy efficient CPU frequency for a system response to user input. The chosen frequency for one interaction would not affect the frequency choice for the next if no policy changes occur. One could construct a case where an interaction takes so long due to a low frequency that it would interfere with the user's plan of issuing the next input. This case is, however, not considered for the workloads in this study. This RL problem subclass is called Multi-Armed Bandit Problem.

In this problem setting a regular choice of a fixed number of options or actions is presented to the agent. The agent selects a single action and receives a numerical reward taken from a corresponding probability distribution. Its goal is to maximise the total reward over time by learning which actions are best. The name bandit is according to the colloquial term "one-armed bandit" used for slot machines in casinos. Its theory is often described with the analogy of a single slot machine with multiple levers. For each play of the machine a single lever can be pulled and the reward is the amount of money won by hitting the jackpot. Each lever has a different payout rate and the goal is to maximise earnings by learning which levers work best.

The agent's decision policy is adapted by learning the probability distribution behind each selectable action. By maintaining an estimate of observed action values the agent can decide which action is best. Like a full RL problem, it faces the decision of whether to exploit current knowledge of observed rewards or explore if other actions might lead to higher rewards. If the probability distribution of single actions changes over time it is considered non-stationary and constant learning is required.

2.4.2 Summary

Reinforcement Learning describes a problem setting where an agent tries to learn which actions to take to achieve a goal. Actions are taken based on the state of the system's environment. A resulting numerical reward is evaluated and used to learn how well the executed action for the given state leads toward the desired goal of maximising the total future reward. A problem subclass of RL is the Muli-Armed Bandit Problem where actions are independent of each other and the agent only makes decisions for a single kind of situation. A major challenge exists in finding a good balance between exploring which actions work best and exploiting the knowledge learned so far to gain optimal rewards.

Optimising DVFS energy efficiency for distinct user interactions in a mobile workload can be classified as a non-stationary contextual bandit problem. The agent needs to find optimal frequency settings from a fixed number of available core frequencies for reoccurring and independent user interactions. Frequency dependent system performance or energy consumption of a distinct interaction can change over time (see Chapter 7 for details). Hence, the developed DVFS agent must constantly adapt its learned behaviour. Publications on RL driven DVFS techniques related to mechanisms

developed in this thesis are presented in Section 3.3.2.

2.5 *Android* Open Source Project

The *Android* operating system developed by *Google* is currently the mobile OS with the highest market share worldwide. According to *Gartner* [1] it held a share of 86.2% of all smartphone sales worldwide in the second quarter of 2016. *Android*'s sources are available to the public via the *Android Open Source Project* (AOSP) [118]. It allows developers to get an insight in *Android* functionality and modify, build and deploy the source for their own purposes. Due to its accessibility and broad application world wide it is a great platform for mobile research and will therefore be used for experiments conducted in this thesis.

All experiments and modifications are based on *Android* Jelly Bean version 4.2.2 with underlying *Linux* kernel 3.4.0. In Chapter 6 a heuristic is developed to detect the ending of a system response as perceived by the mobile device user. It relies on a statistic which indicates when the screen content changes. This statistic is recorded by observing the *Android* framework component *SurfaceFlinger*. *SurfaceFlinger* is responsible for combining all visible surfaces of all running applications to a final frame buffer image which is eventually displayed on the screen. To provide background information, it will be described in more detail in the next section.

2.5.1 *Android* Graphics

The *Android* framework offers two ways for application developers to render images to the screen: With *Canvas2D* or *OpenGLES*. *Canvas2D* is a 2D graphics API which is most commonly used by developers to render their applications. *Canvas2D* operations are translated into *OpenGL* operations which allows hardware acceleration using a GPU. Application developers also have the option of using *OpenGL* commands directly with the *OpenGLES* interface.

Both methods render images to a graphic buffer called a "surface". Surfaces are provided by buffer queues which are at the centre of *Android*'s producer and consumer based graphic pipeline. A buffer queue relays surfaces between image stream produc-

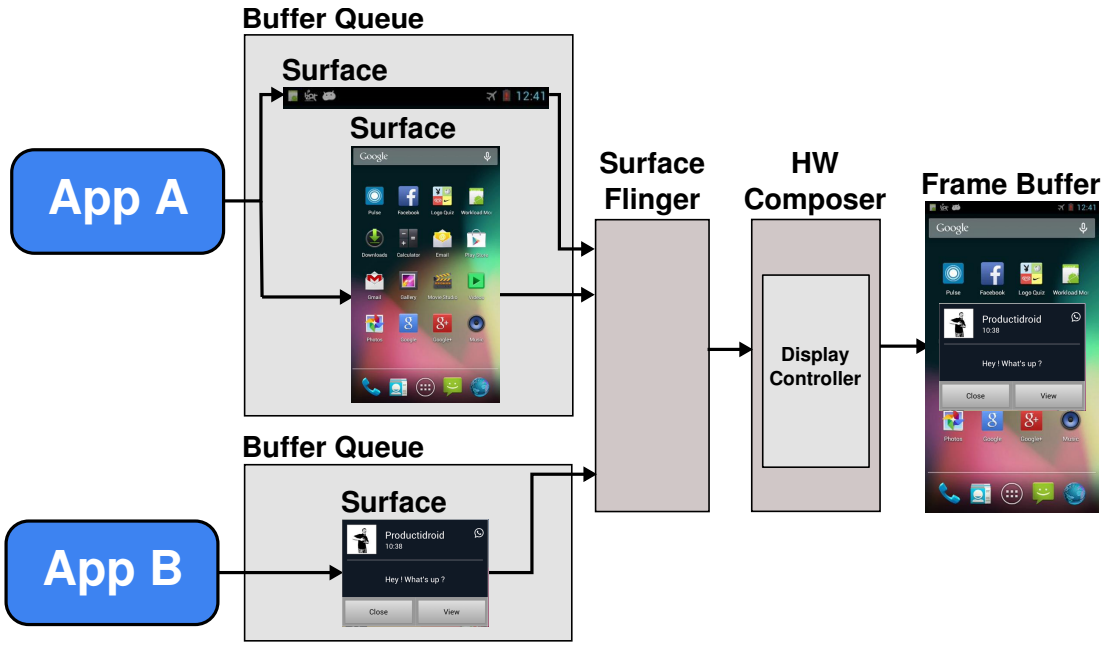


Figure 2.4: *Android SurfaceFlinger* component combines drawable surfaces of active applications to the final frame buffer image presented on the screen.

ers such as *OpenGLES* or *Canvas2D*, and image stream consumers such as *SurfaceFlinger*[1] for final rendering.

The graphic pipeline is depicted in Figure 2.4. On its left side multiple applications are displayed with one or multiple surfaces to be rendered to the screen. Initially, the Window Manager component connects to *SurfaceFlinger* using a buffer queue and requests a surface to draw on. All generated image content is then produced and consumed using this connection. The Window Manager also provides meta information on how surfaces are to be positioned. *SurfaceFlinger* receives all surfaces to be rendered and uses *OpenGLES* and the Hardware Composer to create the final image to be displayed on the screen. The Hardware Composer is a hardware abstraction layer for the display hardware. Its implementation is provided by the device vendor. It decides which way is best to compose images for the given hardware and passes those information to *SurfaceFlinger*.

As soon as an application has finished drawing desired content to a surface, it sends a signal to indicate that buffer production is finished. The buffer is enqueued into the buffer queue connecting the application with *SurfaceFlinger*. *SurfaceFlinger* is notified about new content, acquires it and composites all current surfaces to the final

[1]Even though *SurfaceFlinger* is consuming most surfaces, other consumers exist as well. One example is the camera application which consumes a camera image preview stream and displays it.

screen image. *Android* makes sure that drawing, surface composition and presenting final frames are synchronised with display device frame boundaries to deliver a consistent frame rate.

For the heuristic developed in Chapter 6, *SurfaceFlinger* was instrumented to relay information on its current activity to a message trace. Every time *SurfaceFlinger* is notified about new content and starts composition, a trace message is written to a message stream. This message stream is evaluated by the heuristic to get information about screen changes.

2.5.2 Summary

AOSP is a project providing developers with access to the *Android* operating system sources. It allows developers and scientists to analyse and modify the OS. AOSP includes *Android*'s graphics framework which application developers use to render their applications. No application can render directly to the screen. Instead they render to graphic buffers which are passed on to *SurfaceFlinger*. This component then composites all visible buffers and creates the final image. *SurfaceFlinger* sleeps until it is notified about new content to be rendered. This notification is trapped and used as indicator for screen changes in Chapter 6.

2.6 CPU Frequency Governors

In this thesis a CPU frequency scaling technique is developed for the *Linux* based *Android* OS. It is evaluated against the current standard on this mobile platform. Hence, this section will provide background information on current mobile DVFS techniques.

CPU frequency scaling is implemented in the *Linux* kernel. The corresponding infrastructure is called *cpufreq*. Frequency scaling parameters can be exposed to user space for configuration with the */sys* file system interface. They are located at */sys/devices/system/cpu/*. Like other kernel functionality, algorithm implementations are provided in kernel modules which can be loaded and unloaded as required. In *Linux*, frequency scaling algorithms are called CPU frequency governors. Multiple governor modules are provided by default. The most common ones on mobile devices are listed here:

Powersaver This governor fixes the CPU to the lowest available frequency.

Performance This governor fixes the CPU to the highest available frequency.

Userspace This governor does not implement any frequency scaling policy itself. It serves as a shell for scaling algorithms written in user space. Such programs can use exposed *cpufreq* parameters to monitor and set frequency levels manually.

Ondemand This governor is usually activated by default on device startup. It scales frequency with CPU load. When the load goes up, it jumps to the highest frequency and ramps down when the load declines. Its original algorithm [119] is shown in Figure 2.5.

```
1   for every CPU in the system
2     every X milliseconds
3     get utilisation since last check
4     if (utilisation > UP_THRESHOLD)
5       increase frequency to MAX
6
7     every Y milliseconds
8     get utilisation since last check
9     if (utilisation < DOWN_THRESHOLD)
10      decrease frequency by 20%
```

Figure 2.5: Original algorithm of *Ondemand* governor.

The sample rates *X* and *Y* are functions of the transition latency needed by the underlying hardware to switch between frequencies. *UP_THRESHOLD* and *DOWN_THRESHOLD* specify two percentage values of CPU utilisation and indicate were the governor takes action. *MAX* refers to the maximum available frequency. Most governor parameters such as thresholds can be configured via the */sys* file system interface.

Conservative This governor works the same way as *Ondemand* with the difference that it does not jump to the highest frequency upon detecting increased load. Instead it takes intermediate steps and gradually increases the frequency. This approach is designed with the goal of being more energy efficient. However, it also reduces system responsiveness due to the slower frequency increase.

Interactive This governor also scales frequency with CPU load. In contrast to *Ondemand*, however, it is designed to be more responsive by scaling the frequency up

faster and keeping it high for longer [120]. When the user starts interacting with the device, *Ondemand* would monitor CPU load for several milliseconds before raising the frequency. *Interactive* is designed to detect CPU context switches out of idle and applies only a short utilisation sampling of under 10 ms. This way, frequency is raised much quicker after the CPU came out of idle and the system responds faster. This approach, however, causes the CPU to spend more time in higher frequencies than with *Ondemand* which can lead to higher energy consumption.

2.6.1 Summary

DVFS strategies developed in this thesis are directly compared against *Ondemand*, *Interactive* and *Conservative* and indirectly against *Powersaver* and *Performance*. The last two are considered for experiments by using *Userspace* to fix the CPU to the highest and lowest available frequency manually.

Chapter 3

Related Work

3.1 Introduction

This chapter will discuss research work closely related to the concepts and techniques developed in this thesis. The first section will present research on automating and benchmarking interactive mobile workloads with a specific focus on the evaluation of SRT durations. The second section will look at DVFS research driven by user perception and machine learning methods.

3.2 Interactive Mobile Workloads

In Chapter 4, a methodology is developed to automatically replay interactive mobile workloads on *Android* systems. Input is recorded automatically from actual users considering realistic workload scenarios. Recorded inputs can be replayed an arbitrary number of times. Furthermore, replays are benchmarked measuring SRT durations for each interaction to quantify QOE. This section will at first present workload automation tools for the *Android* OS developed in different studies. Secondly, publications on interactive mobile benchmarking are discussed and lastly research work is reviewed which considers analysis and optimisation of SRT on mobile devices.

Table 3.1: User input automation tools for mobile workloads.

Tool	Test Generation	Comment
Google Monkey	Automatic	Replaying random user input.
MonkeyRunner	Manual	Test cases are scripted.
Robotium	Manual	Test cases are scripted.
Guitar	Manual	Point and click interface to create test cases. No gestures supported.
Gui Crawler	Automatic	Java GUI element crawler. No actual input replay.
Reran	Automatic	*Linux* subsystem input record and replay.
Mosaic	Automatic	Extension of *Reran* to support multiple devices.
Valera	Automatic	Extension of *Mosaic* to include input, sensor and networking data for record and replay.

3.2.1 Interactive Mobile Workload Automation

Replaying user inputs is a commonly used technique for testing systems to find program errors. Usually, a test case composed of multiple interactions is written by hand and then executed automatically as often as required. In the early years of the 21st century, a trend emerged to automate test case generation for applications with graphical user interfaces (GUI). According to *Memon et al.* [121] GUI instrumentation techniques were developed to record user interactions. This led to test cases that were more realistic than manually crafted ones. Multiple frameworks were developed to record and replay user interactions in desktop or server environments. One example is *Xnee* [122], an automation tool for *Linux*. It records user input and related *X11* events. Another one is the *Java* GUI test framework *Abbot* [123]. It can record user interactions for *Java* applications.

With the arrival of modern smartphones automatic generation and replaying of user interactions was needed on these new platforms too. A summary of tools developed for that purpose can be found in Table 3.1. *Google* provides test frameworks for the *Android* OS that are called *Google Monkey* [124] and *MonkeyRunner* [125]. *Google Monkey* is a straight forward stress tester which generates random user input. The more sophisticated *MonkeyRunner* allows developers to manually generate test cases and run them automatically. The third party *Android* GUI testing tool *Robotium* [125] also allows the manual composition of test scripts and automatic replay. It is, however, tied

to distinct applications and cannot replay arbitrary workloads. *Guitar* [126] extends *Android* SDK's *MonkeyRunner* tool to allow developers to create their own test cases with a point-and-click interface. However, *Guitar* does not support complex touch-screen gestures, e.g. swipe and zoom, or other input devices, e.g. accelerometer and compass. A tool developed by *Amalfitano et al.* [127] extends *Robotium* to generated GUI tests cases automatically by crawling the GUI source code and generating *Java* output. It is, however, not able to capture actual user input.

Probably the most sophisticated *Android* interaction record and replaying tool was developed by the University of California. Their initial version called *Reran* [128] is able to record and replay user inputs by directly accessing *Android*'s underlying *Linux* subsystem. The same approach is taken for the work done in this thesis (see Section 4.3.2). *Reran* was later extended to *Mosaic* [129] which allows device independent replay of user interactions to enable portability between *Android* devices. The most recent version *Valera* [130] enriches user input events with sensor readings such as GPS, camera and microphone, as well as networking transmissions. In so doing, *Valera* removes most sources of non-determinism and allows capturing and replaying of realistic *Android* workloads.

All tools presented here allow the generation and automatic execution of test cases for *Android* applications. Most of them require a manual or semi-manual approach to create test cases. Only *Reran* and its predecessors are able to accurately capture and replay actual user interactions. All of the studies presented above, however, only provide a workload automation tool. They do not use it to compose a representative benchmark to share with the research community. Also, they do not use generated workloads to evaluate and improve system behaviour, in particular interaction response times as seen from a user's point of view.

3.2.2 Interactive Mobile Benchmarking

Recording and replaying of user interactions gives developers a possibility to generate representative workloads to test and evaluate their design choices and runtime heuristics. However, to make research results comparable to other studies, those test cases need to be shared in the form of a benchmark suite. Publications presenting mobile benchmark suites exist, however, current research typically still relies on benchmarks built from mobile applications with little to no interactivity and predefined user inputs.

Gutierrez et al. [131] compare mobile workloads to traditional *Spec* benchmarks w.r.t. their micro-architectural behavior. It uses *BBench*, an automated browser benchmark, to open downloaded web pages, scroll to the bottom of the page and measure performance. This sequence of actions does not require any user interaction. Additionally, three further applications (game, music player, video playback) are evaluated, however, these appear to introduce inaccuracies between test runs as their execution is not automated, but are manually launched.

Huang et al. [132] present a benchmark suite comprising popular applications for the mobile *Android* OS, which are executed on the *Gem5* simulator. These benchmarks avoid user interaction altogether and run with predefined input sets.

Pandiyan et al. [133] also present a mobile benchmark suite comprising video playback, image rendering and internet browsing applications. As before, these benchmarks avoid user interaction and operate on predefined user inputs. Thus, this framework does not support record and replay of interactions and no user perception is evaluated.

Kim et al. [134] introduce a mobile benchmark suite called *LatencyBench* which considers user perceived latencies. They use it to analyse dynamic power management schemes commonly used on modern mobile devices such as the *Ondemand*, *Interactive* or *Conservative* governors. Their analysis context focuses on multiple cores. Presented findings show that CPU load information are not sufficient to drive energy efficient power management in an interactive context. Background tasks could take up a lot of CPU time while the user does not notice them. These results are on par with the governor evaluation in Chapter 5. The interactivity in their benchmark, however, is limited to application startups of a browser application and they do not exploit reported findings to implement improved strategies.

Sunwoo et al. [135] study several existing smartphone benchmarks and applications including *AndEBench*, *CaffeineMark*, *Rl Benchmark*, *Angry Birds*, and *KingsoftOffice* with the aim to measure the performance of the *Dalvik* virtual machine, *SQLite* and the OS. They use the *Gem5* simulator and an *autoGUI* system, which captures user input and subsequently synchronises service of input by evaluating the frame buffer. However, like the other methodologies, this approach targets traditional performance metrics like throughput and delay, which do not necessarily translate into an improved QOE for the user.

3.2.3 Analysing SRT in Mobile Workloads

An abundance of tools and studies exist about capturing and analysing usage behaviour of mobile device users and execution statistics of mobile workloads. Some researchers present automatic logging tools [136–151] which run in the background during user's daily routines, others run experiments in a lab environment [8, 9, 152–162] to generate application profiles. Depending on the particular research goal, some studies collect comprehensive traces [136, 140–142, 144, 145, 147–151, 154, 156, 159] of execution statistics and usage behaviour. Among them are application usage, network statistics, micro-architectural features or location statistics. Others focus only on specific parameters such as energy profiles [157, 158, 160], battery charging behaviour [143], touch behaviour [146], wake lock bugs [161, 162] or networking profiles [155]. Among those studies, some consider system response time characteristics of interactive user input with the goal of improving the user's experience [8, 9, 137–139, 152, 153]. Those studies are most closely related to the work presented in this thesis and will be discussed in more detail.

Song et al. [137] optimise SRT for *Android* application startups to improve the end user experience. They propose a usage pattern-based prefetching scheme for mobile devices to improve application startup time. They state that long application startup times negatively affect the user perceived performance of their phones. Frequent page faults and resulting delay due to slow file IO are identified as the reason for the long application startup. By preloading a user's most frequently used applications, the authors aim to reduce application load times. The focus of this study lies on performance rather than energy efficiency. Also SRT is only considered for application starts.

AppInsight [139] is used to monitor SRT characteristics of *Windows Mobile* applications in the wild. The author's use dynamic binary instrumentation of applications to detect user transactions. A user transaction begins when the user interacts with the UI and ends with the termination of all synchronous and asynchronous tasks triggered by the interaction. The authors identify a critical path in a user transaction as the series of task executions starting at the user interaction and ending with a corresponding UI update. They argue that the length of the critical path directly affects the corresponding SRT as perceived by the user. The reports generated by *AppInsight* are sent to application developers to help them improve performance bottlenecks. This study presents a diagnosis tool and leaves evaluation and improvements of analysed applications to

developers.

PanAppticon [138] is also a diagnosis tool monitoring the critical path between user interaction and corresponding UI update in the field. In contrast to *AppInsight*, however, this tool works on *Android* workloads and tracks user transactions across interprocess boundaries as well as into the kernel by considering system calls. This is done by instrumenting both the *Dalvik* virtual machine executing application code and *Android*'s underlying *Linux* kernel. This study also presents an SRT diagnosis tool and offers no actual improvements.

QoEDoctor [9] uses *Android*'s InstrumentationTestCase API to generate automated UI test cases for commercial *Android* applications such as *Facebook*. They use those test cases to analyse user perceived SRT for specific user interactions. By monitoring the application's UI layout tree they aim to capture the exact user perceived start and ending of an interaction lag. Layout tree evaluation, however, is not generic but needs to be implemented separately for each interaction to be analysed. Their evaluation focuses on networking parameters rather than processor performance settings.

Timecard [153] presents an API for server-based interactive mobile applications. It allows the developers to track the SRT for user interactions triggering server requests. Like *QoEDoctor*, it focuses on UI interactions leading to server communication.

3.3 DVFS

In this thesis a DVFS algorithm is developed which bases frequency decisions on how they affect SRT periods as perceived by the end user. A machine learning technique trains a model to predict frequency selections for lower energy consumption and high QOE. The target platform in this thesis is an *Android* mobile device. Developed techniques are evaluated using an automated interactive benchmark which quantifies QOE by considering the user's point of view. The following section will present research related to DVFS techniques driven by user perception and machine learning approaches.

3.3.1 Perception Driven DVFS

Lorch et al. [163] analyse traces of UI events to derive a heuristic to determine when a UI task completes, which is subsequently used to influence DVFS decisions based on

abstract user satisfaction thresholds. The heuristic determining the ending of a UI task by tracking the termination of threads triggered by the interaction. This ending does not necessarily correlate with what the user actually sees as screen output and therefore cares about. The focus of this work is on desktop systems.

One of the earliest works on using user perception driven DVFS for mobile devices was done in 2004 by *Zhong et al.* [164]. They present a dynamic power management and voltage scaling heuristic which considers the interaction delays in interactive workloads on PDAs. Interaction delay is measured by instrumenting event handlers in benchmark applications' source code used for this study. Their approach requires full knowledge of the underlying sources. As before, the interaction delay ending based on event handler observations does not correlate necessarily with the visible ending the user actually cares about.

Yan et al. [165] aim to improve DVFS based on user perceived latencies in system response time for interactive workloads. They monitor events in the *Linux X11* window system to measure latency and to control the frequency governor. The end of a system response as perceived by the user is determined by tracking the communication between *X11* window client and server. The client responds to user input by requesting a graphics update from the *X11* server. The server sends a corresponding update to the monitor and responds to the client. The time between input and the arrival of this response is considered as user perceived response time latency. Still, it is unclear whether the first reported UI update is the one the user actually cares about or if multiple updates are triggered. Additionally, the focus of this work is on desktop systems.

Shye et al. [7] use artificial neural networks to estimate individual user satisfaction levels from the hardware performance counters. They employ explicit user feedback for training a user-aware DVFS algorithm. The focus of this paper is on DVFS for desktop systems, which do not operate under power/energy constraints and typically run different workloads (in this study, e.g. video playback, *Shockwave* animation, *Java*). User feedback is through questionnaires, which are neither automated nor scalable. Workloads cannot be replayed using a different system configuration.

An extended system using biometric input from the user to control DVFS is presented in *Shye et al.* [166]. Experiments are conducted with real users to investigate how different frequency levels in different scenarios affect biometric input. Again, workloads

are not replayable and mainly involve desktop applications.

Same as in this thesis, *Mallik et al.* [167] use an approach to evaluate user satisfaction based on visual output. They measure the rate of pixel intensity changed over time and use this metric to control the frequency governor. Similar to other studies, they use questionnaires, focus on DVFS for desktop platforms and do not make provisions for replaying workloads with modified settings to evaluate success.

Shye et al. [140] create a logging application that collects usage data in the background. Using a linear regression model, power consumption is predicted for interactive workloads. Their results suggest that the CPU together with the screen dominate the power consumption in mobile devices. A proposed scheme for "slow" screen brightness and CPU frequency reduction delivers mixed results. While brightness reduction appears to be effective, perceived random CPU frequency changes introduce lags in games and videos, which users have found annoying. This approach is not automated, but requires users filling in questionnaires.

Bischoff et al. [8] investigate how QOE metrics can be used to optimise a system's energy consumption. Using an architecture model executed with the *Gem5* full system simulator, they try to find CPU frequency and GPU core count for optimal energy efficiency while maintaining a good QOE for the end user. Their QOE metrics are web page render time in the mobile browser benchmark *BBench* and frame rate of a 3D game benchmark. They discuss optimisation strategies but do not present actual implementations.

Song et al. [168] capture user perceived SRT periods for *Android* workloads by using *Dalvik* virtual machine instrumentation. They measure the period between user input and the last UI thread screen update which was triggered by the interaction. Other than previous work they evaluate this technique by comparing it to screen output recordings and achieve a good correlation. They implement a DVFS algorithm which uses the normal *Ondemand* frequency governor during interaction response times and aggressive power saving techniques in the user-oblivious intervals. In contrast to the work done in this thesis they do not consider a user satisfaction model to provide the right amount of performance to save energy at interaction lag time. Also their test set is limited to 7 applications which run manually crafted *MonkeyRunner* scripts.

Zhu et al. [152] present an energy efficient scheduling algorithm for an *Android* platform with heterogeneous cores which is closely related to the work presented in Chap-

ters 6 and 7. They optimise energy consumption by providing "just enough" performance to meet end user's quality of service (QOS) expectations for executed events. Similar to the approach presented in this thesis, they learn correct performance settings for distinct event handlers by observing multiple samples. They apply a user satisfaction model to decide how much performance settings can be reduced before slowdowns become intolerable. Their method is, however, restricted to mobile web applications running in the *Chromium* browser. Also, they do not consider interaction lag as seen by the user. Rather they derive optimal QOS from event handler latencies which do not necessarily correlate with what the user actually sees.

3.3.2 Machine Learning Driven DVFS

A range of machine learning based techniques was developed by scientists to achieve energy efficient DVFS. Most of them are evaluated on desktop or server systems [169–175]. A few, however, consider embedded systems [176, 177]. The most common approach is to capture micro-architectural features such as CPU performance counters, CPU utilisation or memory behaviour [169–173, 176]. Collected features are used to build static prediction models which are applied online to predict optimal performance settings. This approach works well for the workloads the models are trained for but can encounter problems as soon as unseen workload characteristics are encountered.

Reinforcement learning (RL) can be used as a method of dealing with unseen workloads. The learning system dynamically learns correct behaviour over time and is able to adapt to changes. *Shen et al.* [174] and *Islam et al.* [175] present RL approaches to drive DVFS in desktop environments. Both, however, train their system to achieve good results for batch workloads and do not consider interactivity or SRT.

The learning based DVFS study with the closest relation to the work done in this thesis was done by *Li et al.* [177]. They present a supervised learning based energy management technique for *Android* systems called *SmartCap*. Collecting features such as CPU utilisation, touch behaviour and accelerometer data they build a neural network. It predicts a CPU frequency cap on application granularity. The frequency cap serves to save energy and is chosen by considering user experience feedback collected via questionnaires. They try to overcome the problem of unseen applications by providing a user feedback mechanism to send experience feedback to a server. The server processes the feedback to include the new application by training a new model and

distributing it to clients. Considering the size of modern smartphone applications it is unlikely that performance settings on application granularity are sufficient to achieve optimal energy efficiency results. Furthermore, improving the prediction by relying on user feedback is likely not practical in a realistic scenario.

3.4 Summary

DVFS technologies have been developed for decades. Recent technologies such as heterogeneous processing, however, extend DVFS capabilities by increasing the frequency scale with additional cores. This increases potential energy savings and raises the demand for intelligent DVFS solutions once more. Over the last few years user interaction driven energy saving techniques on mobile devices became a hot research topic. It is a way of gaining significant energy savings from DVFS by considering how a user perceives CPU frequency dependent system performance.

To support development of new tools a methodology is required to benchmark energy savings and QOE for mobile workloads. Workload automation techniques and interactive benchmarks exist but none of them systematically quantifies SRT durations as seen from the user's point of view. Chapters 4 and 5 present such methodology and demonstrate energy saving potential.

Within the last year, studies were published which consider SRT to improve DVFS energy efficiency on mobile devices. They show, however, limitations in the workload they can handle by focusing on browser interactions or only idle periods between interactions. Additionally, non of them systematically compares captured SRT periods to the actual screen output as seen by the user. The DVFS algorithm developed in Chapter 6 and Chapter 7 is evaluated against realistic workloads captured from real users. Also, it identifies SRT periods using screen output data by processing information from *Android*'s display subsystem.

Chapter 4

Benchmarking QOE for Interactive Mobile Workloads

4.1 Introduction

The research goal of this thesis is to improve the energy efficiency of DVFS algorithms for modern mobile devices to prolong battery life. DVFS allows the OS to trade performance for power and energy, and vice versa. The DVFS strategies of the standard *Linux* frequency governors like *Ondemand* or *Interactive* are based on the current load of the CPU. As soon as the load of a core reaches a high-threshold, the frequency is raised and when it falls below a low-threshold, it is lowered again (see Section 2.6). This approach works well for non-interactive workloads to deliver performance when it is needed by the core and to save energy when there is nothing to do.

The results of experiments conducted in this work, however, show that the standard frequency governors often set the CPU frequency incorrectly for interactive workloads. They raise the frequency when the user does not need extra performance – for example, when a background task executes while the user is reading text and is unconcerned how quickly the background task completes. The governors also raise frequencies more than is needed to satisfy the user – for example, humans cannot adequately tell the difference between a task running in ten milliseconds or one hundred milliseconds [11–13]. In these cases, the frequency governor wastes energy. Conversely, the governors will not maintain a high enough frequency for long enough and the user will be irritated, waiting for a task to complete. It is, therefore, critically important to consider

Figure 4.1: Execute an interactive workload and record screen output in a video. Automatically identify interaction lag timings in the video and evaluate them in terms of user satisfaction.

the user's point of view, i.e. how he perceives device output, while evaluating how well a frequency governor performs to achieve the best energy efficiency while at the same time providing a high quality of experience (QOE).

None of the current mobile benchmark suites, however, come with an easy-to-use and deterministic method to evaluate user perception for an interactive workload [131–133]. A classic approach to evaluate user perception is using questionnaires [136, 140]. This is, however, a long and demanding process which requires a lot of experiments with many different users to get a statistically sound result. One could reduce the statistical error by making sure that a user always executes the same chain of interactions with the device for every run through of an experiment. For a human, however, this is not only tedious, but a nearly impossible task, especially as the length of the benchmark exceeds a certain time span.

This chapter will introduce a methodology that enables the recording and replay of custom interactive workloads on *Android* mobile devices, and automatically evaluates the effects of changes to the system in terms of user perception. Figure 4.1 shows the concept of this methodology. To get a clear picture of how the user perceives the system, the methodology's first step is executing an interactive workload and capturing a video of what the screen is showing. In the second step, the recorded video is reviewed and the beginning and end of each interaction lag found in the video are marked. In this thesis the time between user input and the time when the user feels that the system has processed his request is called interaction lag (see Figure 4.2). The HCI research community also calls this period system response time. Section 4.2 will describe the concept of interaction lag in an interactive workload in more detail. Marking the be-

ginning and end of all interaction lags leads to an interaction lag profile that lists the length of all lags the user perceived in the executed workload. This profile enables the comparison of the durations of those lags to another execution of the same workload possibly using a different system configuration.

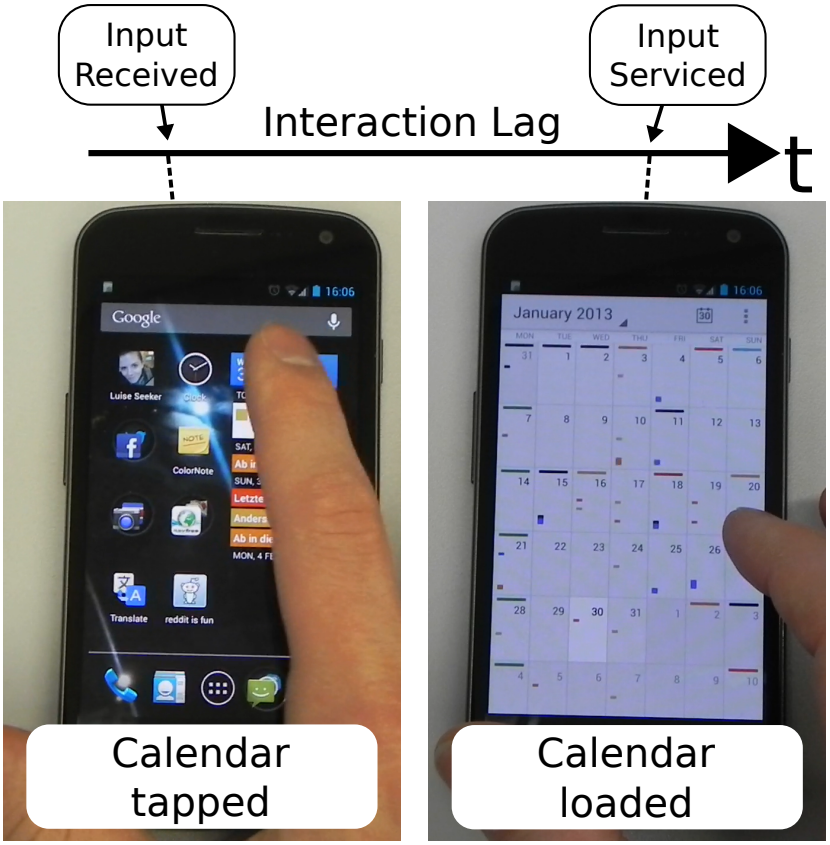


Figure 4.2: Interaction lag is the time between user input and the time when the user feels the input has been serviced by the system. In the example, the interaction lag begins when the user clicks on the calendar and ends when the calendar is fully loaded.

Using the methodology presented in this chapter, a mobile benchmark suite for the *Android* system is generated. This benchmark is composed of 16 realistic interaction scenarios recorded from actual users. It serves as a means of evaluating changes to the system in terms of QOE. In a first step, QOE is measured considering the duration differences between distinct interaction lag profiles. In the next chapter, interaction lag profiles of all captured videos are used to derive a "*user irritation*" metric. It can be used to decide which system configuration was less irritating to the user due to shorter interaction lags.

To demonstrate the benchmark's feasibility, it is executed multiple times while for each execution the CPU is fixed to a certain frequency level. Afterwards, the effect of

varying frequencies on the duration of interaction lags is evaluated. This shows that lag durations shorten the higher the frequency is raised. Durations do, however, not decrease linearly but saturate at a certain frequency. Hence, duration differences of the same lag between the highest frequencies are minimal. This leads to the assumption that it is not always necessary to use the highest possible frequency when executing a lag to satisfy the user. Therefore, energy consumption can potentially be reduced without stretching the length of interaction lags and lowering QOE. This assumption is confirmed in Chapter 5 where an *Oracle* study identifies an optimal frequency profile using the benchmark generated in the current chapter.

4.1.1 Contributions

The contributions of this chapter are:

1. A record and replay mechanism to deterministically replay realistic interactive workloads on the same or another mobile device,
2. a set of interactive mobile workloads used in this study. These form a suite of realistic, repeatable, automated, interactive workloads that can be used by others to compare frequency governor characteristics as well as other system modifications, and
3. automatic detection of interaction lag, based on non-intrusive analysis of video output and device event queues.

4.1.2 Motivating Example

Figure 4.3 shows a short snapshot of how the frequency of the CPU adapts to an input event for two different DVFS governors. The beginning of the user input is marked at point *A*. Point *B* marks the time at which the user would like the input to have been serviced. The thin line represents the frequency using the *Ondemand* governor, while the bold line represents the decisions of an alternative DVFS governor. The *Ondemand* governor uses multiple frequency levels, usually alternating between the highest and the lowest frequency. With full knowledge of the user's perspective, the alternative governor raises the frequency immediately after the input and holds it long enough to ensure that processing is complete before the user is irritated.

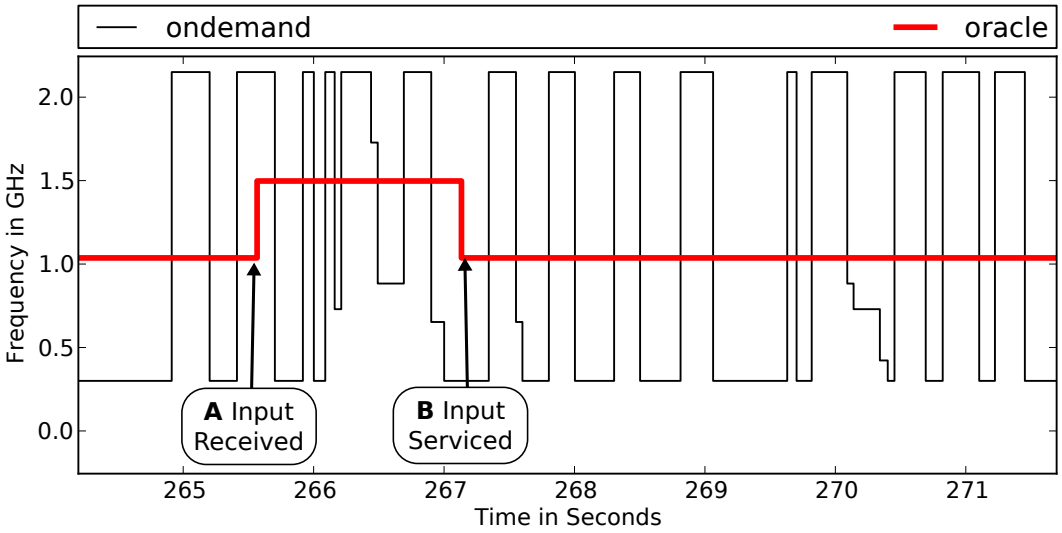


Figure 4.3: Snapshot of the behavior of the *Ondemand* governor and another more energy efficient governor for an interaction recorded using this chapter's novel method. The thin black line indicates frequency choices made by *Ondemand* over the course of the executed workload and the thick red line shows the frequency selections by the more energy efficient governor.

A group of different users was not able to distinguish between the two frequency configurations when confronted with a video of this short example. They were satisfied with the performance of both. But despite this similarity in terms of the user perception of performance, the *Ondemand* governor needs about 30% more energy. In this work two major issues were identified that cause this significant difference in energy consumption:

1. *Ondemand* raises the frequency at times where the user would not notice a difference between a fast or a slow task and therefore would not care. This happens outside of interaction lags.

2. When the user does care, e.g. inside of interaction lags, *Ondemand* overshoots the goal. It raises the frequency higher than necessary to satisfy the user.

Necessary information to avoid these issues can be found when considering the user's point of view of the system. They must include when the user starts interacting with the system, when the user feels that the interaction has been processed and how short the time in between needs to be for a satisfying response time. This interaction lag information can then be used to rank a frequency governor in terms of energy efficiency and QOE.

Since current mobile benchmark suites [131–133] do not offer a way of identifying user interaction lag, the work of this thesis' first technical chapter focuses on creating a new methodology. The identification of beginning and end of user interactions for a mobile workload was initially performed using a straightforward approach. In a first experiment, a camera was pointed at an *Android* Galaxy Nexus and a workload was executed. An analyst would then open the recorded video in a standard video editing tool and step through it frame by frame (see step two in Figure 4.1). Every time he identified a frame as the one where the user submitted an input command, he set a *begin-marker*. Every time he decided that the system now looks like it has serviced the input, he set an *end-marker*. Afterwards, he extracted the number of frames between all markers and had an interaction lag profile. Now two profiles of different executions of the same workload could be compared or overlaid with the corresponding frequency profile to see which frequencies the governor selected.

Unfortunately, the process of marking up a 10 minute video (18000 frames at 30 fps) of a relatively interaction intensive workload takes approximately 4 hours and 15 minutes. This is clearly too costly and inefficient to be of any use and it would be even more so if this process would be used to produce enough data for a thorough study. As an example, the results presented in Section 4.6 and Section 5.6 required 16 different 10 to 15-minute workloads, each one executed for 17 different frequency configurations, with each configuration run 5 times in order to get statistically sound data. All executions translate into roughly 270 hours of video material, for which 6729 hours of markup work would be needed, or about 3 and a half man-years. It is clear that if interactive workloads are to be captured and studied, this process needs to be automated to a high degree. Section 4.3 will present such an automated novel methodology which allows reducing the manual work to a total time of about 2 and a half hours – a speedup by a factor of 2700×.

Next to considering the user's perspective, a second important requirement for improving the frequency governor's energy efficiency is to have realistic and repeatable workloads. For this thesis a group of different users was asked to execute the legacy mobile benchmark suite as proposed in [131]. It consists of playing a Guitar Hero like game, one minute of audio playback, one minute of video playback and a browser benchmark. The browser benchmark automatically loads a web page, scrolls to the bottom and loads the next one. The results of this experiment showed that executing the game manually, as proposed, leads to input event traces with timings that vary by

0.5 to 1 second between multiple runs. The audio and video playback only require a single interaction for the whole workload which is not enough to analyse interaction lag. The browser benchmark is repeatable but none of our users found that it represents a realistic mobile workload since they would not use a device in such a way. A way of recording and replaying actual, rather than artificial, user interactions is needed with millisecond accuracy to get representative workloads. These need to be repeatable without major deviations in order to compare multiple executions. With the technique presented in this chapter, users can create repeatable and realistic workloads as they would naturally execute them.

4.1.3 Overview

This chapter is structured as follows. Section 4.2 will give more detailed information on how an interactive mobile workload is perceived by a user focusing on its visual representation. Section 4.3 will present the novel approach of automatically detecting interaction lag in realistic and repeatable workloads. This is followed in Section 4.4 by a description of the generated workloads using recordings of real user interactions. Experimental setup to execute the workload and apply the methodology to a CPU frequency study is shown in Section 4.5. A presentation of experimental results can be found in Section 4.6, which are summarised and concluded in Section 4.7.

4.2 The User's Point of View

To improve DVFS techniques for interactive mobile workloads it is necessary to know how frequency changes affect system performance as seen from the user's point of view. In this study, the user's point of view refers to the user perceived screen output. In interactive workloads an indicator for performance differences perceptible by the user is the time the system needs to respond to input. As shown by various researchers (see Section 2.3 for details), too long system response times increase the user's irritation, cause him to make mistakes and reduce QOE. Knowledge of when the user feels that the system handles an interaction and when he perceives the system as idle to await the next one are therefore useful to determine when high performance is required and when it can be traded for energy efficiency.

The methodology presented in this chapter makes use of interaction lag durations to quantify a user's QOE with a workload execution. It identifies interaction lags by marking their beginning and end in a video recording of the workload. To provide a clear picture of what interaction lags for *Android* workloads look like, the following section will present an execution example and show how it is perceived by the user. Furthermore, it will point out interaction lag boundaries in the visual representation of the executed workload by decomposing it into two major types of execution periods: *user perceived interaction lag period* and *user perceived system idle period*.

4.2.1 Interactive Workload Example

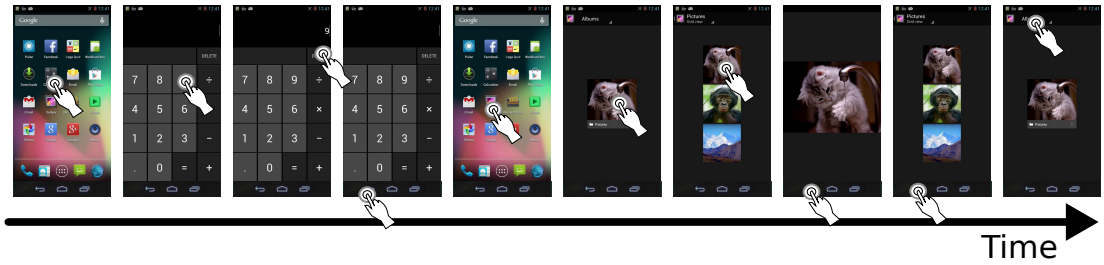


Figure 4.4: Screen output of a short sample of user interactions with an *Android* device. Screenshots along a timeline show the mobile device's screen output for every executed user interaction. A hand symbol marks user touch screen taps. First the calculator is started and used. After returning to the home screen the Gallery application is started to look at pictures.

Figure 4.4 shows the screen output for a few seconds of user interactions with an *Android* device. Execution time spans from left to right. A screen snapshot was taken at every time the user interacted with the touch screen over the course of the execution. In detail, the course of the interactions goes as follows: *Execution starts on the home screen where the user taps on the calculator application shortcut. Once the calculator is loaded the user taps on key nine and afterwards on delete which removes the number again. A tap on the back key closes the calculator and brings the user back to the home screen. Now the user taps on the home screen shortcut for the Gallery application which brings up the Gallery. He taps on the visible album and is presented with its contents. He selects the cat image which is opened and presented in full screen. Two successive taps on the back button bring the user back to the album overview in the Gallery.*

This short interaction example demonstrates what the mobile workloads presented in

this thesis look like. Each interaction is started with the user issuing input by tapping or swiping on the touch screen of the device. The system then reacts to the input and presents the corresponding results. This process is followed by the next input and so forth. Points of interest in a workload are therefore times at which input is issued to the device, i.e. where interaction with the device happens.

4.2.2 Lag and Idle

The time between inputs can be subdivided: After the user issued an input he usually expects the device to react to it, e.g. by starting the Gallery application. Once the Gallery is started he takes some time to observe the presented result, decides on his next interaction and proceeds. *Shneiderman* calls the first period "system response time" (SRT) and the second "User think time" (UTT) [12]. The time after an input can therefore be subdivided into a period where the user waits for the system to respond and a time where he observes the presented result. This concept is depicted in more detail in Figure 4.5.

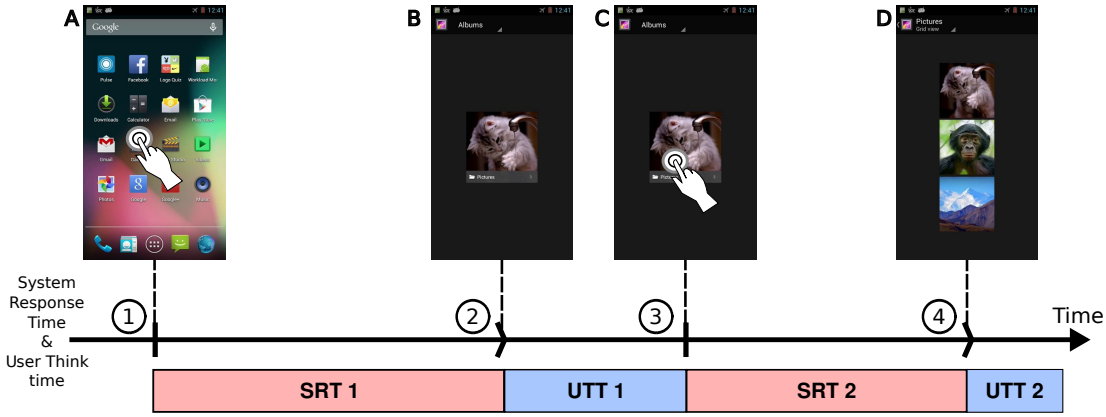


Figure 4.5: Distinct phases of interactive input. System response is followed by a user think period until the user issues the next input. The straight and angled markers show the beginnings of corresponding periods.

In this figure two input events happen which are marked on the time line on the bottom of the figure. At the top of the figure, the corresponding screen output is shown. The first input at ① is followed by a period where the system responds by highlighting the tapped homescreen shortcut, fading in the Gallery application and drawing the albums (SRT 1). At ② with the screen output B the user feels that the system has finished processing the input and observes the result (UTT 1). Once the user decides to proceed, he taps on the album to open it at ③ and the second response period starts.

Now the system fades in the picture contents of the album (SRT 2). At ④, this is again followed by an observation period once all pictures are loaded (UTT 2).

It is important to keep in mind that the subdivision is made from a user's point of view. It is likely that the underlying system activity does not exactly match what the user is seeing presented on the screen. If the user perceived the system to be finished loading the Gallery application, the system could in fact still be loading elements in the background of which the user is not aware. While he is observing presented results and is perceiving the system as idle a task could be issued by a timer to check emails which is serviced in the background.

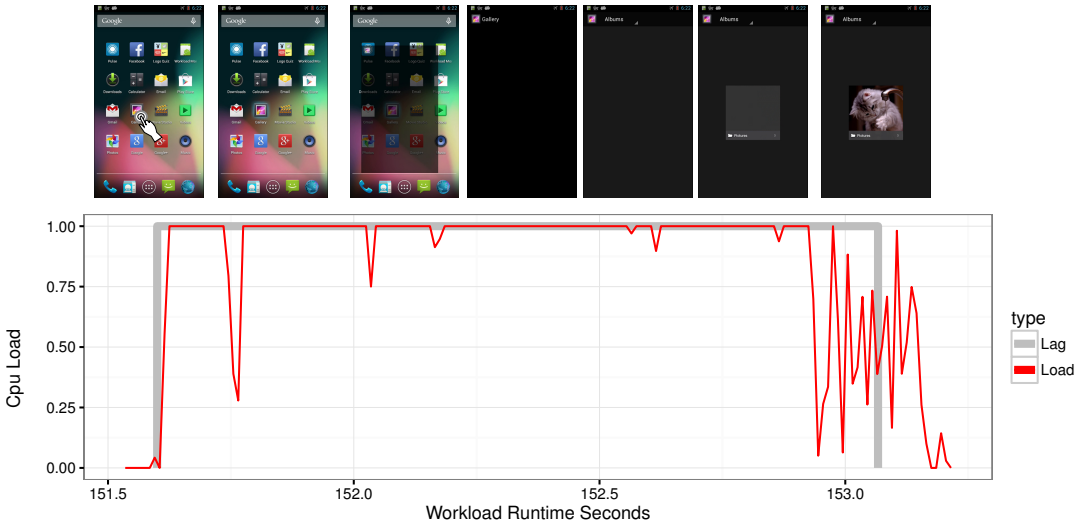


Figure 4.6: User perceived workload compared to activity of the underlying system. The activity is represented by the CPU load during workload execution. A thick grey line marks beginning and end of user perceived lag time and a thin red line indicates CPU load.

Figure 4.6 shows an example of underlying system activity compared to perceived system activity. A single input and corresponding system response is shown in this figure. It is presented from both the user's point of view and the system side. CPU load of a single core CPU is displayed to indicate system activity. The input as indicated by a hand pointer in the first screenshot is again a tap on the Gallery shortcut which opens the application. This time not only screenshots of input events are displayed but also a few in between to indicate what happens during the system response. A thick grey line indicates where the user perceived response starts and where it ends. As soon as input is being issued the CPU load goes up. While it is lower towards the end of the user perceived system response, it does not completely settle to zero. There are also additional spikes during the following observation period. This example

shows that there can be differences in activity depending on the point of view. Hence, a promising DVFS approach is to provide high performance during periods of user perceived activity and saving energy when the user thinks the system is idle.

The following terminology will be used for the remainder of this thesis:

Interaction Lag The interaction lag period is the system response period as perceived by the user. It is the time span where the user waits for the system to finish processing his request after an interaction has been issued. It is therefore called lag.

System Idle The system idle period or simply idle period is the period following a lag. It starts after the user feels that the system has finished processing the previous input and lasts until he issues the next one.

Both interaction lag and system idle period can have various durations depending on the corresponding interaction. If the text of an email is typed in, for each key press both periods would be expected to be short, e.g. in the range of 200 - 500 milliseconds. If an article is being opened and read on the screen the observation period could have a length of several seconds. At the same time, loading the article to the screen is also potentially longer than a simple key press. If a game is started and a loading bar appears the interaction lag could last for several seconds while the system loads necessary components.

The difference between the two periods is that the length of the interaction lag depends on the system while the length of the idle period depends on the user. In the interactive workload considered for the experiments in this thesis, the user would usually wait until the system has finished processing his request before issuing the next input. As long as he observes interaction results and thinks about what to do next the system appears to be idle. Therefore, perceived system performance only depends on interaction lag durations. As human computer interaction (HCI) research suggests, the user has an implied deadline for an interaction lag (see Section 2.3.3). If this deadline is exceeded, the user's impatience grows and QOE degrades. To automatically benchmark QOE for an interactive mobile workload, the methodology presented in the next section captures interaction lag durations.

4.3 Methodology

This section will describe in detail how the methodology for automatic benchmarking of QOE in interactive mobile workloads functions. Firstly, an overview of all steps taken to automate the method will be presented. This is then followed by a detailed description of each involved mechanic.

4.3.1 Automation Steps Overview

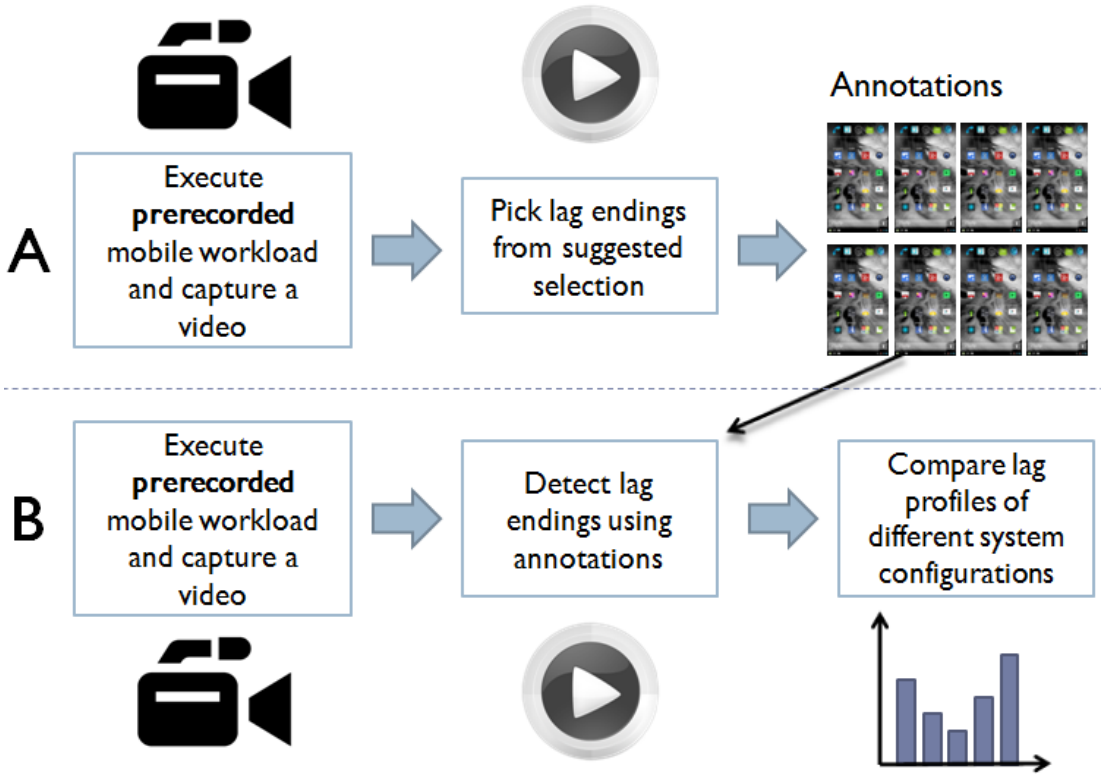


Figure 4.7: This figure shows the automated version of the proposed method to create repeatable and realistic workloads and to evaluate them in terms of user perception. Part A shows how a workload is annotated. This task needs to be executed only once per workload. It produces an annotation database containing an image of the expected ending for each interaction lag. Part B is then fully repeatable for the same workload. Here the annotation database is used to automatically mark up a video of the workload's execution and produce a lag profile.

Instead of executing a workload manually for each run, user inputs are recorded once and then replayed independently. Input events are captured directly from the *Linux* input subsystem, so it is possible to replay them in exactly the same way and with

accurate timings whenever needed. This is done by following the same approach as presented in other studies [128].

Knowing the exact timings for all input events already provides the beginning of each interaction lag. Finding the ending of an interaction lag is automated in the next step. The ending is the time when the user feels that the system has serviced his input. To do this a matcher algorithm was implemented that uses a database of images. These images show for each lag how the expected ending looks like on the mobile screen. The matcher steps through the captured video of the workload execution frame by frame. Starting at each lag beginning, it finds the corresponding lag ending by comparing each frame to the expected image. With recorded inputs, the matcher and the database, the workload is repeatable and its interaction lag evaluation fully automatic.

The image database is created by *annotating the workload*. Annotating a workload means selecting an image for each interaction lag that shows how the mobile screen looks like when the end user feels that the system has serviced his input. This needs to be done only once, after which the workload will be reusable time and again. This process was made easy for the workload creator by automating most of it as well. In the manual markup method in Section 4.1.2 the video analyst had to look at all frames in the video that follow the begin frame to identify a corresponding end. Instead of looking at all frames, the workload creator now only has to look at a small selection of frames which already have a high potential of being the correct one. These potential ending frames are automatically selected by a suggester algorithm for each lag. The workload creator only needs to pick the right one. This takes on average only a couple of seconds per interaction lag. The image the workload creator picked is then added to the workload's image database which is later used by the matcher.

Figure 4.7 shows the automated version of the method derived from the former concept in Figure 4.1. Part A shows the annotation step which needs to be executed only once. Here a prerecorded workload is run and a video of it captured. The suggester algorithm then presents a selection of potential lag ending frames for each lag beginning and the workload creator picks the correct ones. Part B is fully repeatable and can be executed an arbitrary number of times for the same workload with different system configurations or even different mobile devices. This is under the requirement that the initial system state of the device is always the same. Again, a prerecorded workload is run and a video is captured. The matcher algorithm now automatically finds the corresponding lag ending for each lag beginning using the annotation database and

produces a lag profile. This profile can then be compared with profiles of other video evaluations of the same workload in terms of user perception.

4.3.2 Automatic Record and Replay of Interactive Workloads

In order to accurately record and replay a workload executed on an *Android* phone, input events are captured directly from the *Linux* input subsystem. This system has a standard interface for handling the input provided by various peripheral devices and sensors. On a mobile, those devices would be, for example, a touch screen, hardware buttons, a light sensor, etc. The hardware driver format of the input events captured by the single device drivers is converted into a standard input event format. All incoming events for each active device can be accessed via the */dev* file system interface. The input event interface for the touchscreen of the Galaxy Nexus, for example, can be found at */dev/input/event1*.

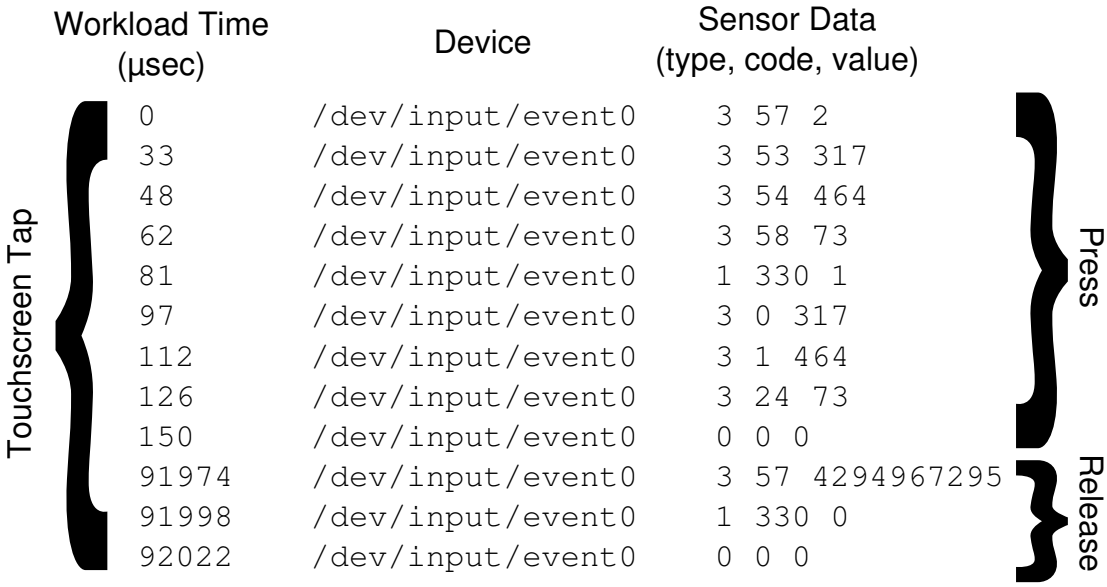


Figure 4.8: Input sensor data recorded by the *GetEvent* tool for a single tap input on the touchscreen. Recorded sensor data is split into type, code and value and gives information on touch coordinates, size and duration.

A single touch is composed out of multiple input events as shown in Figure 4.8. The three displayed columns show elapsed workload time (already translated from system time since startup), the input device's id (touchscreen in the given case) and sensor data carrying various information. The first number specifies the type of event like a key or button press, relative motion or absolute motion. The second number specifies a code of which button or axis is being manipulated and the last number specifies the actual value.

Record *Android* provides a tool called *GetEvent* which is a front-end to reading the */dev* input event interface. When a workload is recorded, this tool is used to capture executed input events together with exact timestamps. The recording process needs no external hardware support, it can be executed on users' devices, while it is carried with them about their daily business.

Replay *Android* also provides a tool called *SendEvent* which is a front-end for writing to the */dev* input event interface like the corresponding device driver would. Unfortunately, this tool is very basic and does not provide enough functionality and performance to replay recorded event traces accurately. Therefore, for this study a custom event replay agent was implemented. This agent knows the input event trace which was recorded and replays it with accurate timings. It is based on *SendEvent* and adds required accuracy.

4.3.3 Capturing Screen Output

Figure 4.9: A video of the mobile screen output is captured by recording an HDMI signal with a video capture device like the *Elgato Game Capture HD* [178].

Figure 4.9 shows how a video of the mobile device screen is captured. Rather than using a camera, the direct screen output is captured via HDMI. This way image artefacts are avoided which would significantly complicate the process of comparing video

frames with each other. Many modern mobile devices have either a MINI-HDMI socket or support the MHL or SLIMPORT protocol which returns an HDMI signal over the MICRO USB port. The HDMI signal is forwarded to a video capture device like the *Elgato Game Capture HD* [178] which decodes it and sends it to a desktop or laptop via USB. There, an application records the signal and creates a video file with a standard format.

4.3.4 Semi-Automatic Markup of Workload Videos

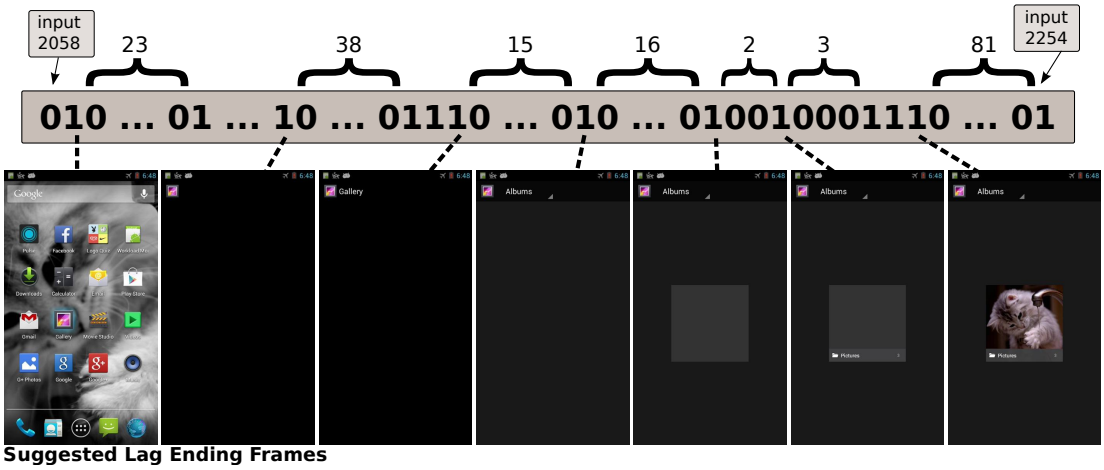


Figure 4.10: The suggester algorithm maps successive video frames to a sequence of ones and zeros. A zero is assigned to a frame that looks equal to its predecessor and a one to each frame that differs from it. Each one preceding a zero is then suggested as potential lag ending since it marks the beginning of a period of still standing images.

As mentioned in Section 4.3.1 a semi-automatic process is used for marking interaction lag beginnings and endings in a workload video. Instead of looking at all possible frames, a suggester algorithm picks out a small selection of frames that have a high potential of showing the correct lag ending, i.e. the state of the system where the user feels that the system has finished servicing his input command. The workload creator can then quickly pick the correct one for each lag out of the provided selection.

Figure 4.10 shows an example of how this suggester works for user input leading to an interaction lag. The interaction being executed is a click on the Gallery shortcut on the home screen. This interaction will cause the Gallery application to start. The state considered the end of servicing the input is when the Gallery is completely loaded and showing the image album overview. The small box on the top left side shows that an input occurs at video frame 2058 and the small box on the top right shows the next

input at frame 2254. The ones and zeros in the long box show the suggester's inner representation of each frame in the current video snippet. The suggester algorithm compares successive frames and assigns a zero to a frame that is equal to its predecessor and a one to a frame that is different. For visibility, chains of zeros are summarised using curly brackets.

The single number one above the first image indicates a frame change where a blue highlight is drawn around the home screen shortcut of the Gallery after tapping on it. This is followed by a short pause and therefore a string of zeros. Then the application window fades in with successive frames constantly changing. A headline appears and changes from "Gallery" to "Albums" and finally an album item is drawn by popping in its background, name and title image in succession.

The images on the bottom of Figure 4.10 are all suggestions made by the algorithm. In the workload used for this study's experiments, the point at which users determine the end of processing an input is always the last of some number of changing frames, i.e. the last one in a chain of ones. The end point is never during a period of unchanging frames, i.e zeros. The algorithm suggests an end image for each one preceding a zero. That way a frame is suggested if it is the first of a period of still standing images (a range of zeros following a one). There are always periods of still standing images which are picked out as the potential ending of an interaction lag. The still period can be very short, for example with on-screen keyboard input, or very long, for example when reading an e-book.

In Figure 4.10 multiple suggestions appear while the Gallery loads up single elements of the final screen one by one. Loading the Gallery takes about 200 frames at the lowest CPU frequency (about 6 seconds at 30 fps) and leads to 8 to 10 suggested images. The number of frames the workload creator has to look at is therefore reduced by a factor of 20. When a workload contains long periods without screen updates the reduction in the number of frames can be much larger.

The suggester can be configured for each interaction lag to make the process of picking a frame more convenient and faster. If, for example, a blinking cursor is producing a long string of suggestions, the suggester can be set to allow a certain amount of pixel difference between frames. If a small animation prevents the suggester from finding still standing images, a mask can be applied to hide it. The amount of zeros following a one can be specified to control the expected length of a still period. If it were set to 30

in the given example, the number of suggestions would be reduced to 2 and the correct one would still safely be caught. A workload creation GUI allows these settings to be explored and tuned easily.

4.3.5 Detecting Lag Endings Using the Annotation Database

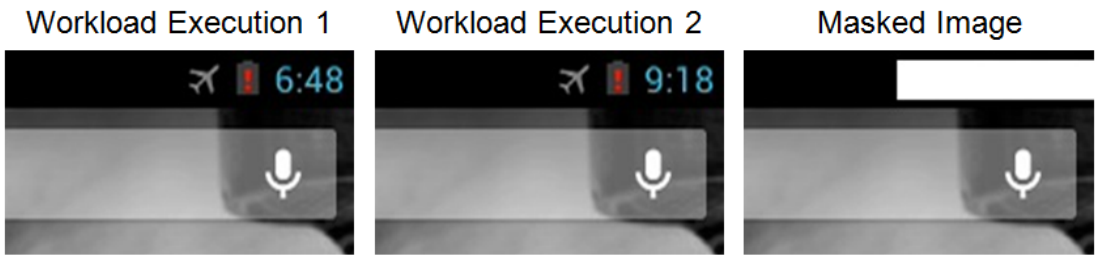


Figure 4.11: Parts of the images being compared can be masked out to handle a certain degree of non determinism between workload executions. In the example, the clock is masked out so the matcher can find the required ending image for different workload executions.

Now that an annotation database was produced containing an image of how each interaction lag ending is expected to look like, it can be used to mark up videos of any further execution of the same workload. The matcher algorithm steps through the video frame by frame and looks for a lag beginning according to input timings. As soon as a time is reached where an input was issued, it picks the corresponding lag ending from the annotation database and compares all following frames with that image until it finds a match. The time between beginning and end is then saved in a lag profile.

In order to find ending frames in new videos of the same workload, it is important that the executed input events stay in sync with the state of the system. For example, if a button needs to be pressed, the system must have reached the spot where the corresponding screen is visible. This will then lead to the expected ending image and will allow the next input to be placed correctly. This can become an issue for random contents like advertisement pop-ups or randomly generated levels in games. In the workload recorded for this study, these contents were avoided where possible. However, the method is able to handle a certain degree of non-determinism with the following techniques.

When annotating the workload in the markup process, it is possible to specify additional information for each lag. For several lags it is necessary to specify an image mask to be used by the matcher. If, for example, the system clock needs to be masked

out when comparing images or a random advertisement looks different for every time a workload is executed (see Figure 4.11). It could also happen that the user input leads to an interaction which ends up on the exact same screen as where it was started. For example sending an email could pop up a loading bar which disappears again after the email is sent. The suggested lag ending therefore looks like the beginning. In this case the workload creator can specify that the matcher should look for the second occurrence of the required image. The GUI makes it easy for workload creators to change mentioned parameters and to both design custom masks and to apply standard ones. Such additional information is saved together with the image in the annotations database and helps the matcher to successfully find the lag endings in a video. With this system, the workloads can be replayed and analysed fully automatically.

With a methodology to generate workloads and automatically annotate interaction lags in place, a benchmark workload can be created. The following sections will present details on the generated workload. Furthermore, experiments will be presented which were conducted to test the feasibility of the lag annotation method. Experimental results show how varying the CPU frequency affects lag durations in an interactive workload.

4.4 Generated Workload

To generate a representative mobile workload 16 people were asked to use a mobile device with the recording system installed as presented in Section 4.3.2. Each used the device for about 10 to 15 minutes. No further instructions were given, beyond asking that they "exercise the software". Their interactions with different applications and widgets were recorded on the device (see Table 4.1). Before users were allowed access, the target platform was reset to a known state to ensure that the recorded workload could be rerun from that same state later. Table 4.1 gives an overview of which activities users executed. In total, user activity time of 190 minutes was recorded. According to a recent study of *eMarketer* [179] the average Americans used their mobile phones for 174 minutes over a 24 hour period. The generated workload therefore covers more than a day of normal phone usage.

Figure 4.12 shows an input classification for all datasets recorded from all users for this study. Left hand side bars show gestures classified as tap inputs and swipe inputs.

Table 4.1: An outline of the main activities users were executing in the recorded workload.

Executed Activity
Image manipulation with Gallery application.
Playing a logo quiz game.
Pulse News widget and multimedia text messaging.
Pulse News application.
Movie Studio video creation.
Calculator calculations.
Writing emails with Gmail application.
Google Translate and Dictionary application lookups.
Writing and editing notes with the Google Keep application.
Reading and managing articles with the Pocket application.
Browsing descriptions of sights with the Stay application.
Checking weather updates with the Weather Pro application.
Browsing recipes with an Indian recipes application.
Browsing music albums and tracks.
Creating and editing contact entries.
Using the telephone dialler application.
Browsing podcasts with the BeyondPod application.
Using the stop watch application.

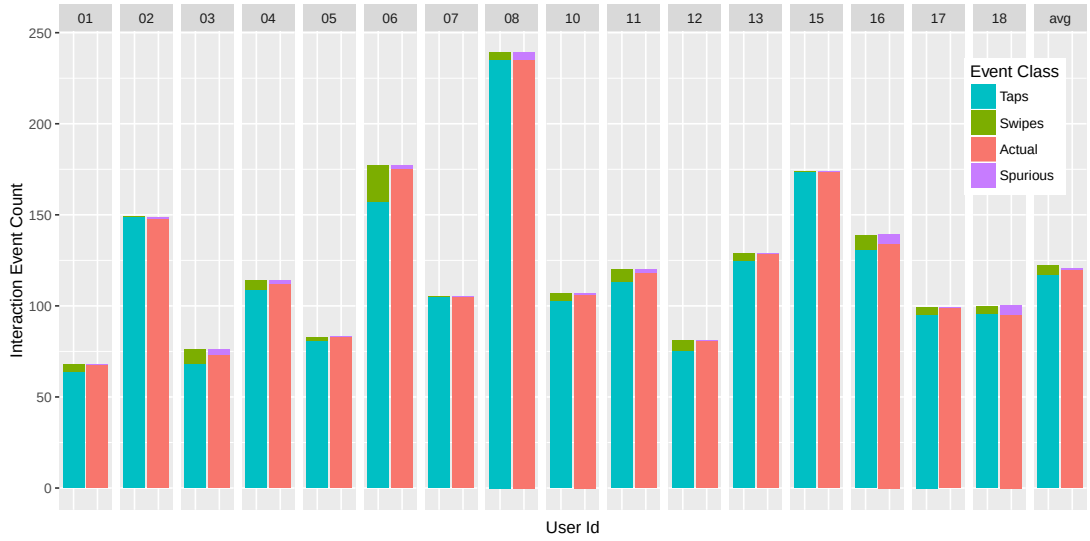


Figure 4.12: The graph shows an input classification for all input traces of the workload recorded from participating users. Left hand side bars show gestures such as taps and swipes and right hand side bars show spurious and actual lags.

The tap inputs are dominating due to the nature of the recorded workloads. In total 1935 input events were recorded. Out of which 4% were swipes and the remainder taps. The method can currently only handle interaction lags that result from tap inputs

and simple swipes like changing the home screen. More complicated gestures such as complex swipes, pinches or drags mostly lead to interaction lags that require a different analysis which is generally described as Jank lags and considered for future work[1]. Those interactions were excluded from user generated content. Also interaction events are excluded at the beginning and the end of recording periods where the interface to activate and deactivate recordings was used. This leaves a total of 1852 interactions.

Right hand side bars show the number of inputs identified as actual lags and inputs that were spurious lags. It can happen that an input event does not lead to a reaction from the system. If the user, for example, taps next to a button or a settings menu is not supported for a certain application, the system will just ignore the input. Therefore, these inputs are considered as spurious lags and ignored. They make up a share of 1% of the total input event count.

4.5 Experimental Setup

Figure 4.13: Qualcomm Dragonboard APQ8074 (image source [181])

[1]Jank lags appear in long animations such as video playback or quick scrolling [180]. They are perceived as a "stutter" in the frame rate. In those cases the system is unable to process new frames fast enough and drops some. This leads to a user experience which appears unsmooth.

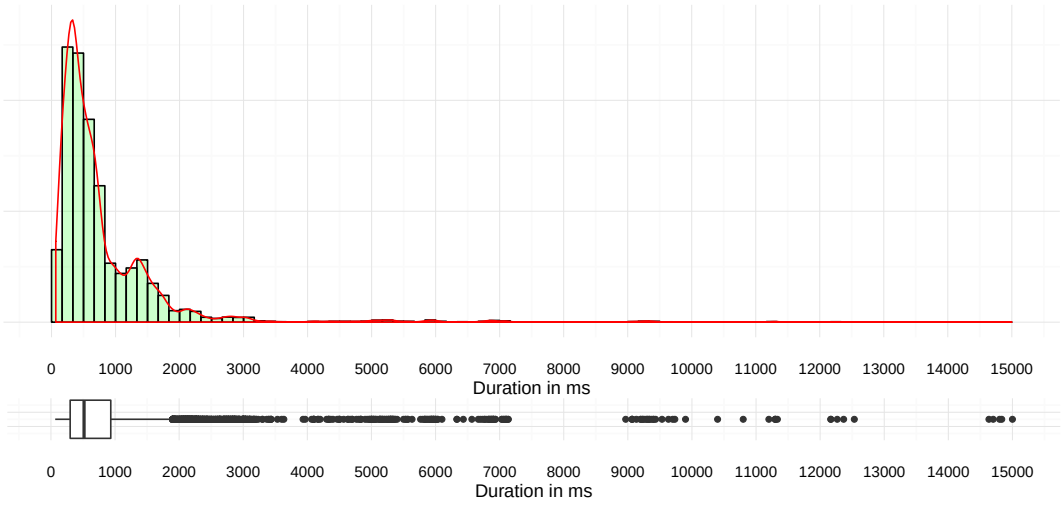
To analyse the length of different interaction lag periods for varying performance the Qualcomm Dragonboard APQ8074 was used in this study (see Figure 4.13). The Dragonboard is based on the Snapdragon 800 quad core processor. It has the same underlying architecture as the Google Nexus 5 mobile phone but allows easier access to connectors and interfaces to measure energy and modify various parts of the system. The board runs *Android* Jelly Bean version 4.2.2 with *Linux* kernel 3.4.0. For the experiments all cores are switched off except one. This is done to avoid statistical noise from load balancing between different cores when the CPU frequency is used to control performance.

In order to gather sufficient data points to analyse the influence of performance on interaction lag durations, the generated workload was executed for each available core frequency. During those executions the CPU is fixed to the selected frequency over the whole runtime. The Snapdragon 800 processor used for this study allows 14 different frequency points ranging from 0.3 GHz up to 2.15 GHz. To reduce the statistical error, this process was repeated 5 times for the entire workload. The standard error is calculated across all 5 iterations. Altogether the generated workload was executed $5 \times 4 = 70$ times.

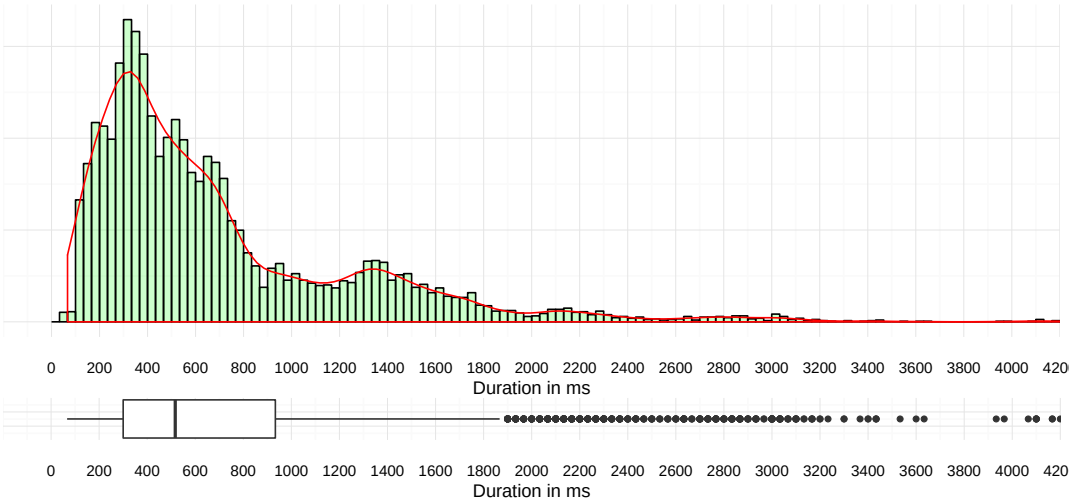
4.6 Experimental Results

Figure 4.14a shows a histogram demonstrating the distribution of interaction lag durations over the whole recorded workload. The frequency configuration chosen for these graphs is a medium frequency of 1.04 GHz. The bulk of the interaction lag durations falls within 300 and 933 milliseconds with a median of 516 milliseconds. The bucket size chosen for the histogram is equal to the duration of 5 frames. The videos were recorded at 30 fps which gives a single frame a duration of 33.33 ms. Figure 4.14a shows a zoomed-in version of the data. The x-axis is limited to 4 seconds to give a more detailed overview of the lower part of the lag duration distribution. Here the bucket size is reduced to a single frame. This is the minimum possible resolution for lag durations due to using the automatic lag duration detection method described in Section 4.3.4.

To better understand which interactions users executed in the workload Figure 4.15 shows a human readable duration categorisation of interactions from the generated



(a) Five frame bucket size.



(b) One frame bucket size and zoom in on the first 4 seconds.

Figure 4.14: Histogram plot of interaction lag durations for a medium frequency configuration of 1.04 GHz. The box-and-whisker plot in the lower part of each sub figure extend from lower to upper quartile values, with a line at the median. The whiskers show the range of the lag duration at 1.5 times the interquartile range, while flier points are those past the end of the whiskers.

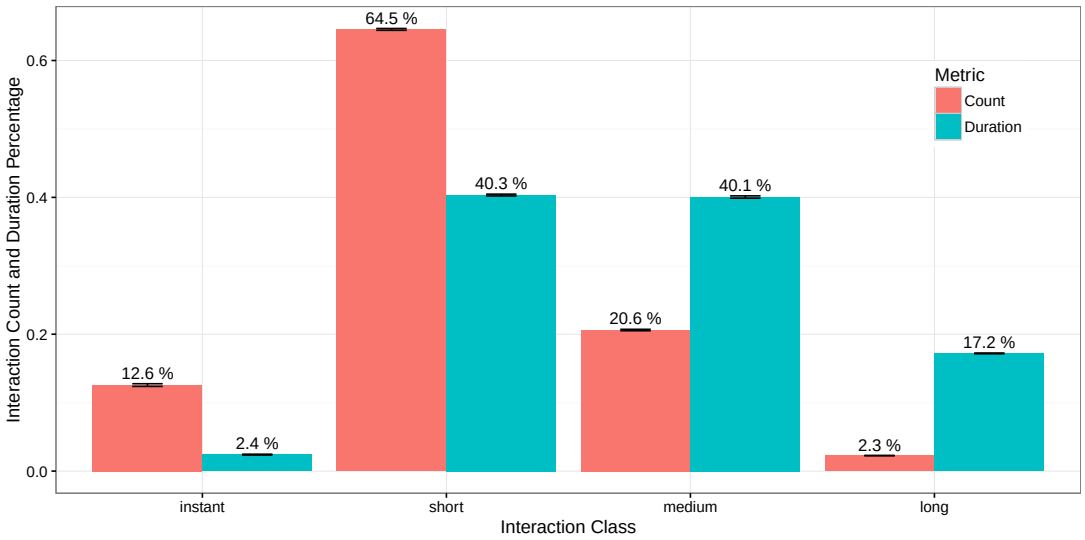
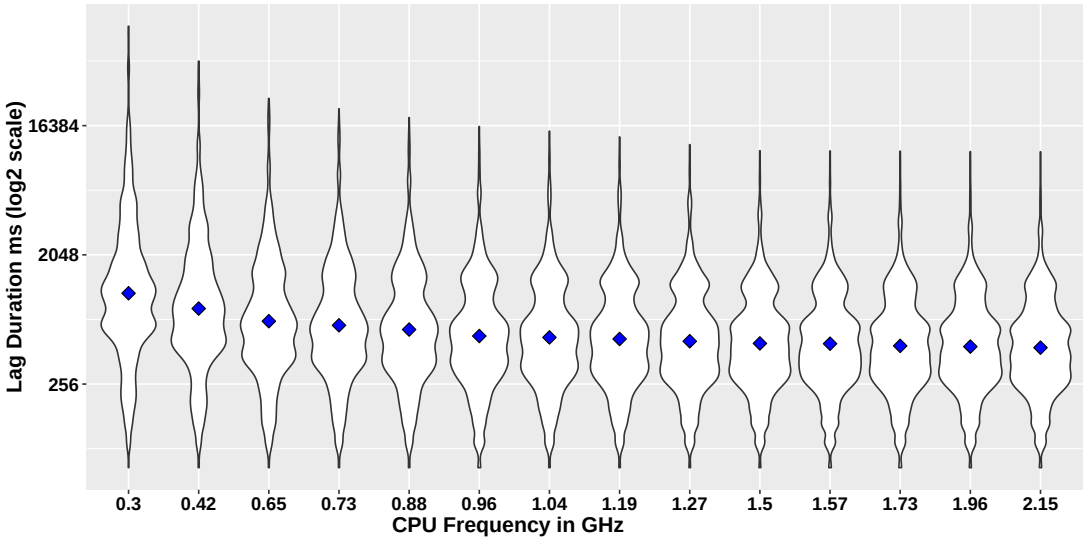


Figure 4.15: Shares of interactions according to duration categories. The left hand bar represents the share of the total interaction count for the given category, the right hand bar shows the share of the total workload time. Error bars show the standard error which is calculated over all 5 iterations of the workload execution.

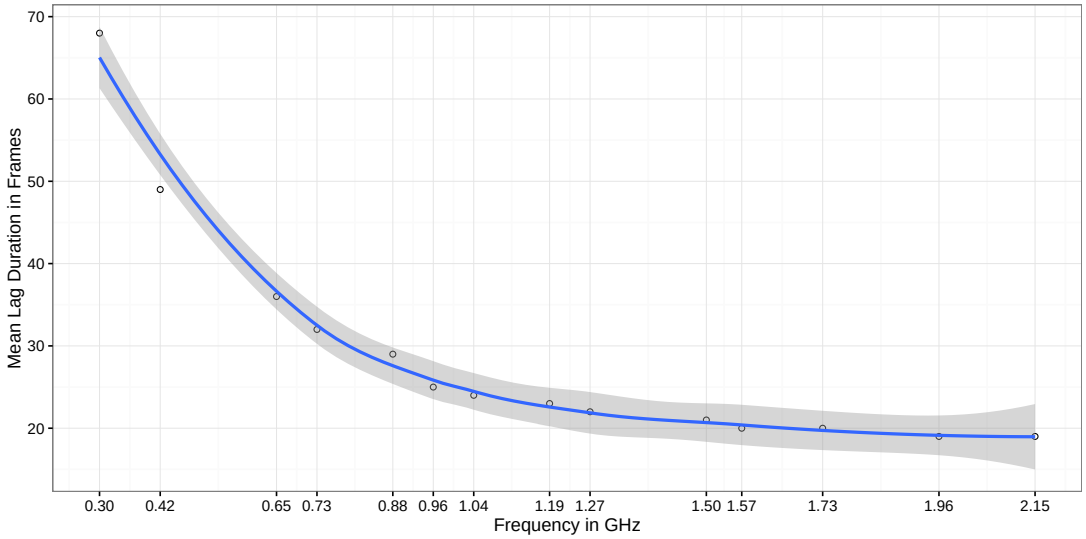
workload. All durations are taken from a workload execution on medium frequency of 1.04 GHz. The four categories are instant response (50ms - 200ms), short lag (200ms - 1s), medium lag (1s - 3s) and long lag (3s and above). Bars on the left indicate the share of total interaction count for each category and bars on the right indicate the share of total interaction duration of the corresponding category. With 64% most interactions are short which can be explained by looking at the users' activities in Table 4.1. They spent a lot of time with typing-heavy applications such as writing emails, using dictionaries, drafting notes or playing a quiz game which requires putting in guess words. Typing a letter on the on-screen keyboard usually has a short response time. A few long response times can be seen where the interaction lag time goes up until 15 seconds. Among them are starting up applications that need to load content from the sdcard such as Pulse News, the Indian recipe or the Stay tourist application. Also saving modified images to disk and encoding videos from the Movie Creator are among them. When looking at the distribution of total interaction lag duration instead of total interaction count for the given classification: short and medium durations have about the same share of 40% each. Most of the rest is taken by long tasks with 17.2% and only 2.4% are used by instant responses. The latter ones are interactions such as ticking a checkbox or highlighting an article in the Pulse News widget.

Figure 4.16a shows a series of violin plots for the lag durations of all available CPU

frequencies the experiment was executed for. The y-axis is on a log_2 scale to increase the visibility of outliers. The overall shape of each plot is about the same with a bulk around the centre and thinning out towards longer and shorter lag durations. It is, however, more or less vertically stretched out depending on frequency height. The higher the frequency, the shorter the lags and therefore the less stretched out the violin plot. But also the difference in vertical span becomes minimal at around 1.5 GHz going towards higher frequencies.



(a) Violin plots of the lag durations for all available frequency configurations with y-axis on log_2 scale. The dot in each plot marks the corresponding mean.



(b) Trend line for all total lag duration means in frames for all available frequencies (a frame has 33.33 ms). The grey ribbon around the trend line indicates the 95% confidence interval for points on the fitted curve.

Figure 4.16: Distribution of lag durations over the generated workload.

Figure 4.16b displays this trend more directly. Here, the lag duration mean in frames of each available frequency configuration is shown. An approximation curve is fitted to all data points. It shows that lag duration mean decreases non linearly from lower to higher frequencies. It follows an exponential shape and starts to saturate around 1.5 GHz. Above 1.5 GHz the difference to the fastest possible frequency of 2.15 GHz is no more than 2 frames. According to HCI research [11–13] the end user does not notice a duration difference below 150 to 200 ms, i.e. about 5 to 6 frames. This means that a frequency lower than the fastest possible can be used for a large number of interaction lags without causing a performance difference perceptible by the user.

4.7 Conclusion

In this chapter a novel methodology was introduced to automatically benchmark user QOE for interactive mobile workloads. It does so by identifying interaction lag periods following user interactions. Lag profiles generated for two executions of the same workload can be compared in terms of interaction lag times. Differences in lag times indicates which execution provided higher QOE. In the next chapter this method will be improved by introducing a single metric mapping multiple interaction lag durations to a single user irritation value. By recording and replaying user interactions with an *Android* device a real interactive workload was built and used for a feasibility study. In this study system performance was changed by fixing the CPU to different frequencies. The effect of performance changes on interaction lag durations was analysed. With the experiments conducted in this chapter it was shown that the length of the interaction lags as they are perceived by the user can be influenced by modifying the CPU frequency. More importantly, it became apparent that there is room for selecting frequencies lower than the highest possible one for interaction lags without reducing QOE to an amount noticeable to the user.

The presented methodology and the generated workload will be used in the following chapter to evaluate how well current DVFS governors are performing in terms of energy efficiency and QOE. Results of this chapter will show were they waste energy by selecting too high frequencies. Furthermore, an *Oracle* will be developed which is able to find a frequency profile for a workload which results in the lowest possible energy consumption whilst not showing perceptible performance loss to the user.

Chapter 5

QOE Driven DVFS *Oracle* Study

5.1 Introduction

The previous chapter introduced a novel methodology to identify interaction lag in interactive mobile workloads. Additionally, a representative workload was generated by recording interactions from actual users. By replaying the workload and applying the lag marker methodology a mobile system can be benchmarked in terms of user perceived lag timings. These two components are necessary to accomplish this thesis' goal of optimising DVFS energy efficiency for mobile workloads by considering the user's point of view. The optimisation approach taken in this study is driven by the observation that current standard DVFS techniques on mobile devices use too high frequencies where the user would not notice a performance difference. These observations are presented in this chapter by benchmarking lag durations of standard frequency governors and evaluating where they choose energy inefficient frequencies.

To identify those inefficiencies frequency selections of standard governors are compared to a perfect *Oracle* baseline. The *Oracle* always selects frequency levels which result in maximum energy savings whilst showing no perceptible performance reductions compared to running at the fastest frequency. To quantify perceptible performance differences, a user irritation metric is introduced. This metric functions by considering a threshold for the duration of each interaction lag in a workload. If lag duration exceeds the threshold the user becomes increasingly irritated. If a lag duration stays below the threshold, performance differences are imperceptible. The accumulated time by which all lags in an executed workload exceed corresponding thresholds

serves as overall irritation score aka QOE score. For each lag, the *Oracle* picks frequencies which stay below the threshold whilst still achieving energy savings. In between lags the *Oracle* picks the least energy consuming frequency. Here, performance is irrelevant since the system appears idle to the user (see Section 4.2).

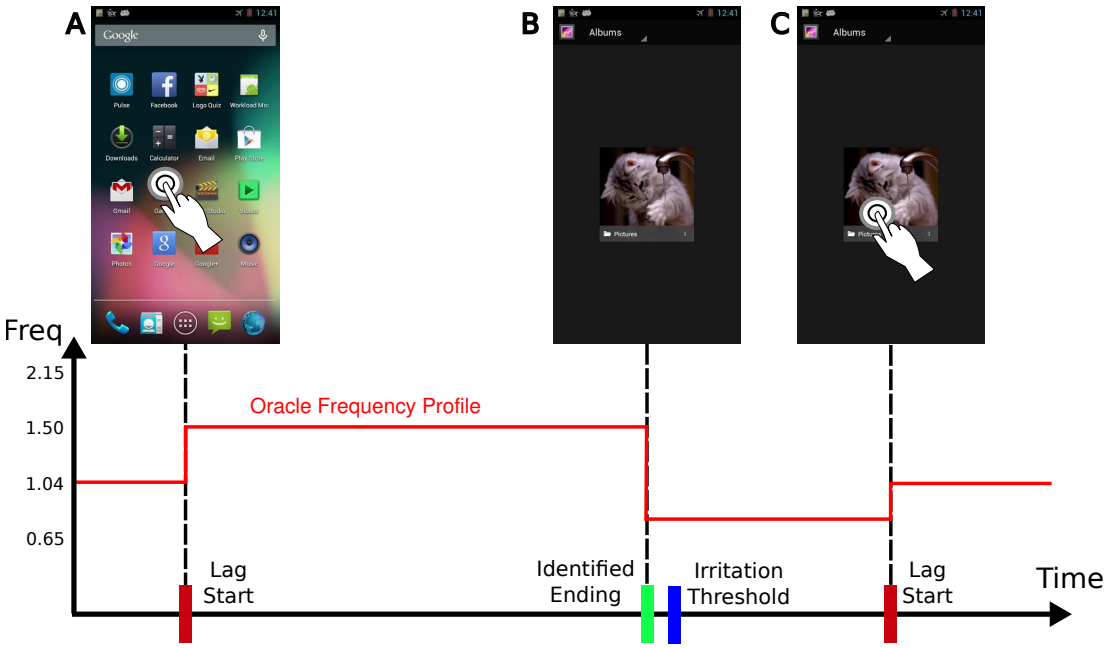


Figure 5.1: This figure shows what an *Oracle* frequency selection for an interaction example looks like. These frequencies result in maximum energy savings and have no perceptible performance impact on lag execution time. At lag time the lowest frequency still below an irritation threshold is chosen. At idle time the frequency with the lowest energy consumption for that period is chosen.

The concept of the *Oracle*'s frequency selection is depicted in Figure 5.1. This figure shows two interactions which are indicated by the markings on the time axis and the screenshots in the upper part. Screenshot A and C show the screen output of lag beginnings. A finger icon marks where the user touched the screen. The first touch starts the Gallery application which finishes loading on screenshot B. The graph on the bottom of the figure shows frequency selections made by the *Oracle*. For the lag period between A and B a frequency level is selected which causes a lag duration shorter than the lag's irritation threshold (marked on the time axis). At the same time the frequency is not the highest possible and energy is preserved. Using perfect knowledge of all possible frequency configurations, the *Oracle* selects a frequency for the following idle period which results in the highest energy savings for that period.

The standard *Android* frequency governors evaluated against the *Oracle* baseline are *Ondemand*, *Interactive* and *Conservative*. Energy efficiency and user irritation results

show that neither of them performs particularly well. Up to 32% energy savings are possible whilst delivering a user experience that is better than that provided by the governors. Furthermore, results show that it is possible to save 45% energy with performance that is indistinguishable from permanently running the CPU at the highest frequency.

5.1.1 Contributions

The contributions of this chapter are:

1. A metric derived from interaction lag profiles to classify user irritation for a particular workload and,
2. a study on how the measurement methodology from Chapter 4 can be used to provide a baseline frequency profile with maximum energy-efficiency for interactive workloads. This is done whilst maintaining the same or even improving system responsiveness compared to three standard governors.

5.1.2 Overview

Section 5.2 will introduce the lag duration based metric used to give an irritation score to workload executions. This is followed in Section 5.3 by a description of the power model used for calculating energy consumption of an executed workload. Section 5.4 will provide a detailed description of how the *Oracle* selects frequencies for interaction lag and idle periods. The experimental setup to evaluate frequency governors against the *Oracle* is described in Section 5.5 and corresponding results are discussed in Section 5.6. Lastly, Section 5.7 will summarise and conclude the chapter.

5.2 Irritation Metric

The methodology presented in Section 4.3 generates interaction lag profiles for each annotated video of an executed workload. A lag profile lists the lag length for each interaction lag in the video. This profile is used to compare the lag durations of multiple executions of the same workload for different performance configurations. To quantify lag durations of a single workload execution a new irritation metric is introduced which

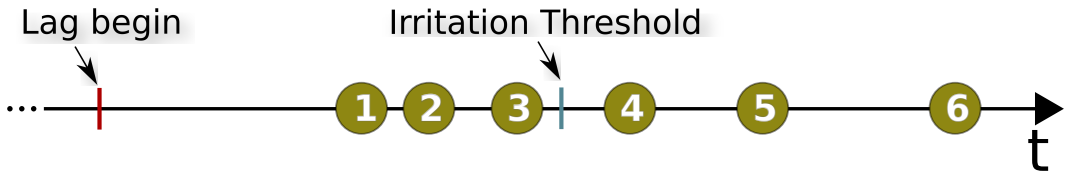


Figure 5.2: This figure shows the timeline of a single interaction lag. Each circled number stands for the lag ending of a specific system configuration. Each lag length that stays below the specified Irritation Threshold does not count as irritating and for each lag length that exceeds it a penalty is applied.

assigns an irritation score. Different executions of the same workload can then easily be compared in terms of user irritation and thereby QOE.

Figure 5.2 shows a timeline for a single interaction lag. The lag's beginning is marked and each circled number stands for a lag ending. The endings were found in experiments where the same interaction was executed with different CPU frequencies and therefore the lag duration differs. ① marks the ending of the fastest frequency and ⑥ the ending of the slowest. Before applying the metric it needs to be configured by setting an *Irritation Threshold*. If the lag length is below this threshold, it does not count as irritating to the end user, if it is above, an irritation penalty is given. The penalty is the amount of time by which the lag duration exceeds the threshold. The metric is an accumulation of the penalty for each lag in the workload, i.e. the total amount of time a user is irritated by too long lag times.

For the experiments conducted in this chapter, the Irritation Threshold is set independently for each lag. This is done while creating the workload as described in Chapter 4. When picking the interaction lag ending from the suggested selection, the workload creator can choose the threshold from a standard HCI taxonomy, for example the ones presented in Section 2.3. He can also apply a custom model or specify each Irritation Threshold individually.

As explained in Section 4.2, the system idle period does not need to be considered when calculating the user's irritation. Since the length of the idle period depends on the user's decision to continue interacting with the device, there is no correlation to system performance. An irritation tied to the system performance can therefore not be applied. There might be other QOE factors next to performance which apply to the idle period like screen brightness or frame rate, but they lie outside the scope of this study. The irritation metric proposed here assumes an irritation of zero for each idle period.

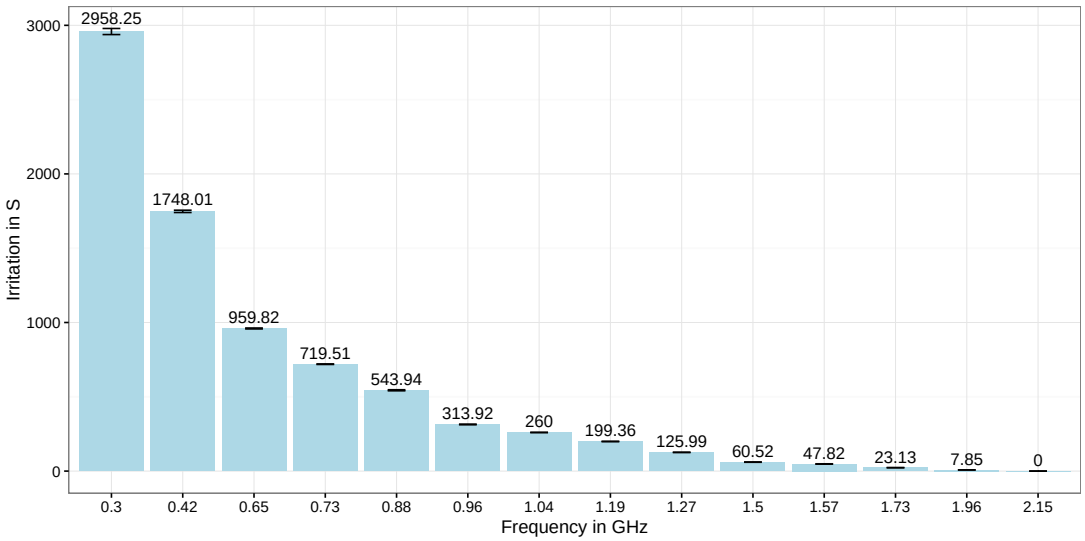


Figure 5.3: Irritation metric applied to workload execution for different CPU frequencies. The CPU was always fixed to the corresponding frequency for the entire execution of the workload. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

To get an idea of how the metric is applied, Figure 5.3 shows the metric's results for the generated workload executed for each available frequency configuration. During execution the CPU is fixed to the corresponding frequency. To calculate the standard error the workload is executed 5 times for each frequency. The irritation thresholds used for this experiment is set to 110% of what the fastest frequency could achieve for each lag. Setting irritation thresholds this way was chosen for two reasons:

1. It is assumed that the fastest frequency is fast enough to handle most interactions with acceptable QOE. Additionally, the fastest frequency is taken as upper cap since a higher frequency is not possible on the given device. With irritation thresholds configured in that manner, the metric therefore is only applicable when looking at a single type of device. When using the metric for a different case, for example, to compare multiple devices, customised irritation thresholds can be configured.

2. According to HCI research [13] presented in Section 2.3.4, a performance difference that is meant to be noticed by the user needs to be as high as 20% of the original value. With 110% of the fastest frequency, this boundary is not breached.

The y-axis of Figure 5.3 shows irritation in seconds. According to the metric this

can be seen as the total amount of time the user was irritated over the course of the workload while waiting for the system to respond. The bars indicate that total irritation degrades the faster the frequency becomes and the fastest frequency is by definition not irritating.

5.3 Power Model

In order to calculate the energy efficiency of a workload with the currently selected frequency profile, a power model is used. When executing a mobile workload with the methodology presented in Chapter 4, a data monitor collects various execution statistics. Among them is data on CPU busy time. Using the power model and the time the CPU is busy during workload execution, the energy consumed by the CPU can be calculated. The model is generated by executing a CPU intensive micro benchmark for each core frequency and measuring overall system power. Afterwards, the idle system power is subtracted to get dynamic core power for each frequency. Figure 5.4 shows an example application of the power model. It displays the energy consumption in kilojoules for multiple executions of the same workload. Displayed energy values do not include the static component of the CPU's power dissipation, i.e. leakage power P_{leak} (see Chapter 2.2.1 for details). Same as in Figure 5.3 the CPU was fixed to the corresponding frequency for each run and the standard error was calculated over 5 iterations of each execution.

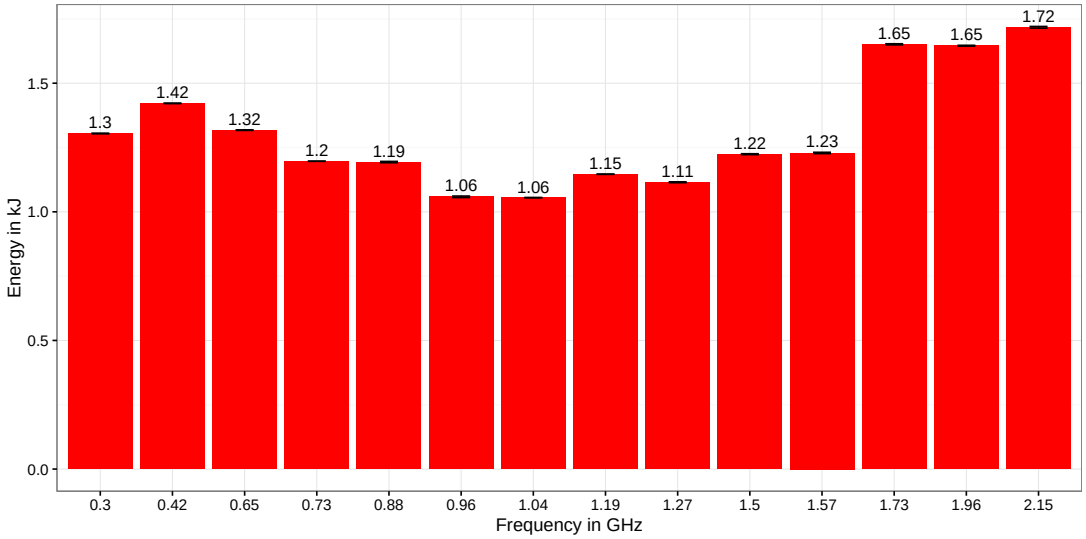


Figure 5.4: CPU energy consumption of workload execution for different CPU frequencies. The CPU was fixed to the chosen frequency for each execution. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

5.4 Frequency Selection *Oracle*

The *Oracle* is used to provide a baseline frequency profile which shows the highest achievable energy savings for the benchmark workload whilst having an irritation score of zero. It has perfect knowledge of energy consumption and lag duration caused by each frequency at each point of an executed workload. This knowledge is collected by exhaustively executing the benchmark workload for each available core frequency. Lag durations and energy consumption of each lag and idle period are measured. An offline algorithm then calculates the *Oracle* frequency profile:

Lag For lags the *Oracle* considers the most energy efficient frequency which results in a lag duration still below the irritation threshold which was specified during workload creation.

Idle For idle periods following each lag irritation is zero per definition (see Section 5.2). Hence, the *Oracle* chooses the least energy consuming frequency for that period.

The following sections will explain frequency selections for lag and idle periods in more detail.

5.4.1 Selecting a Lag Frequency

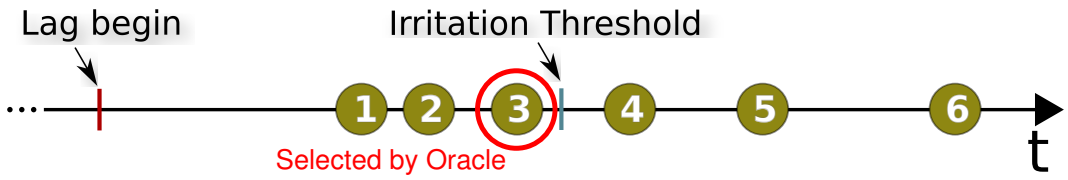


Figure 5.5: This figure shows the timeline of a single interaction lag. Each circled number stands for the lag ending of a specific frequency. Each lag length that stays below the specified Irritation Threshold does not count as irritating and for each lag length that exceeds it a penalty is applied. The *Oracle* selects the least energy consuming frequency which is still below the threshold (circled).

Consider the interaction lag example in Figure 5.5 where each number stands for the lag ending of a certain frequency configuration. ① marks the ending of the fastest frequency, ② the ending of the second highest and so forth. To construct the *Oracle* profile the most energy efficient frequency is picked for each lag that is still below the

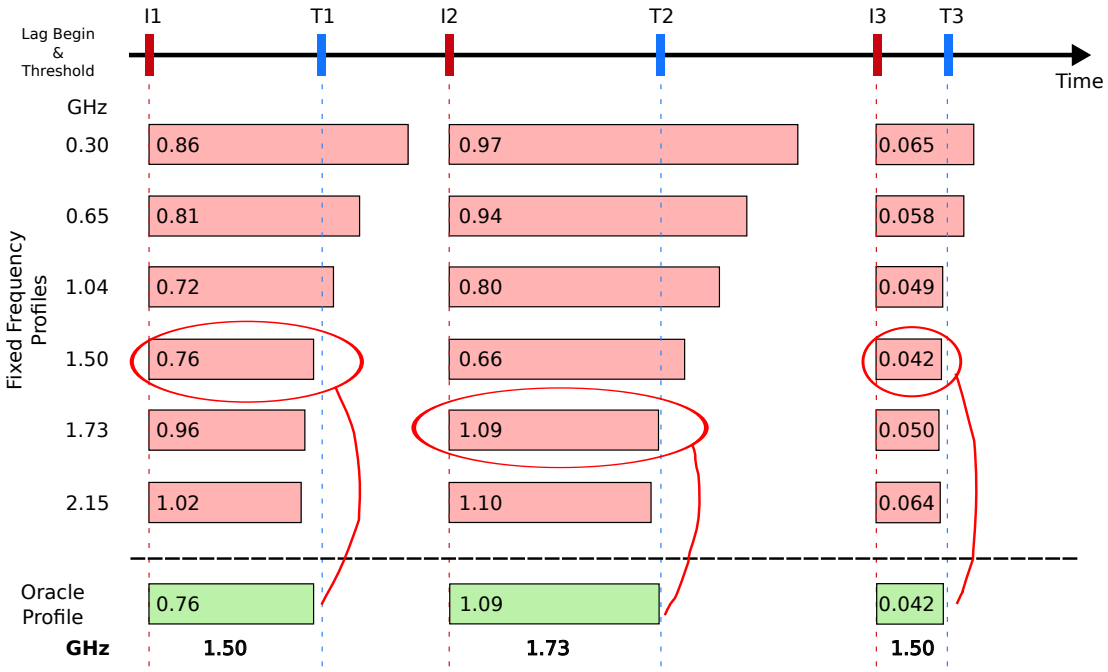


Figure 5.6: Pictogram showing how the *Oracle* algorithm selects lag frequencies for an example of three interactions. For each lag (square red box) frequency profiles are displayed showing lag duration (box size) and energy consumption in Joules (number in box). Frequencies 0.30 GHz - 2.15 GHz are available on the CPU for this example. The timeline on top marks interactions (I) and corresponding irritation thresholds (T). The boxes on the very bottom below the dashed line indicate which frequencies were chosen by the *Oracle*.

irritation threshold which was specified during workload creation. In Figure 5.5 this would be frequency ③.

Figure 5.6 shows how the *Oracle* selects lag frequencies for an example of three successive interactions. At the top of the figure a timeline is displayed where interaction events are marked with (I) and corresponding irritation thresholds with (T). Below this timeline, rows of frequency profiles are displayed. Each row stands for an execution of the interaction example while the CPU was fixed to the corresponding frequency. The 6 displayed frequencies range from 0.30 GHz until 2.15 GHz. Each frequency profile indicates collected execution data by showing coloured boxes. The size of a box stands for the lag duration while the number inside the box indicates the lag's energy consumption in Joules. All boxes in this figure start at input times (vertical dashed line below timeline ticks marked with *I*). Their duration differs depending on frequency and some exceed the irritation threshold (vertical dashed line below timeline ticks marked with *T*) while others stay below. On the very bottom of the figure below the horizontal dashed line, the *Oracle*'s frequency selection for the three lags is displayed.

The higher the frequency for a single interaction, the shorter the corresponding box, i.e. lag duration. Energy consumption does not scale linearly with frequency level. This can also be observed when looking back at the power model in Figure 5.4. The lowest frequency does not necessarily mean lowest energy consumption. As discussed in other research, predicting the effect of DVFS decisions on performance is not a straightforward endeavour for a realistic workload task mix (see Section 2.2.2). Having perfect knowledge of all execution statistics, however, the *Oracle* can pick the least energy consuming frequency which leads to a lag duration still below the irritation threshold. In the presented example this results in 1.50 GHz for the first interaction, 1.73 GHz for the second and 1.50 GHz for the third. Now that lag frequencies for the three interactions are fixed as well as corresponding lag durations, the *Oracle* proceeds with frequency selection for the idle periods in between lags.

5.4.2 Selecting an Idle Frequency

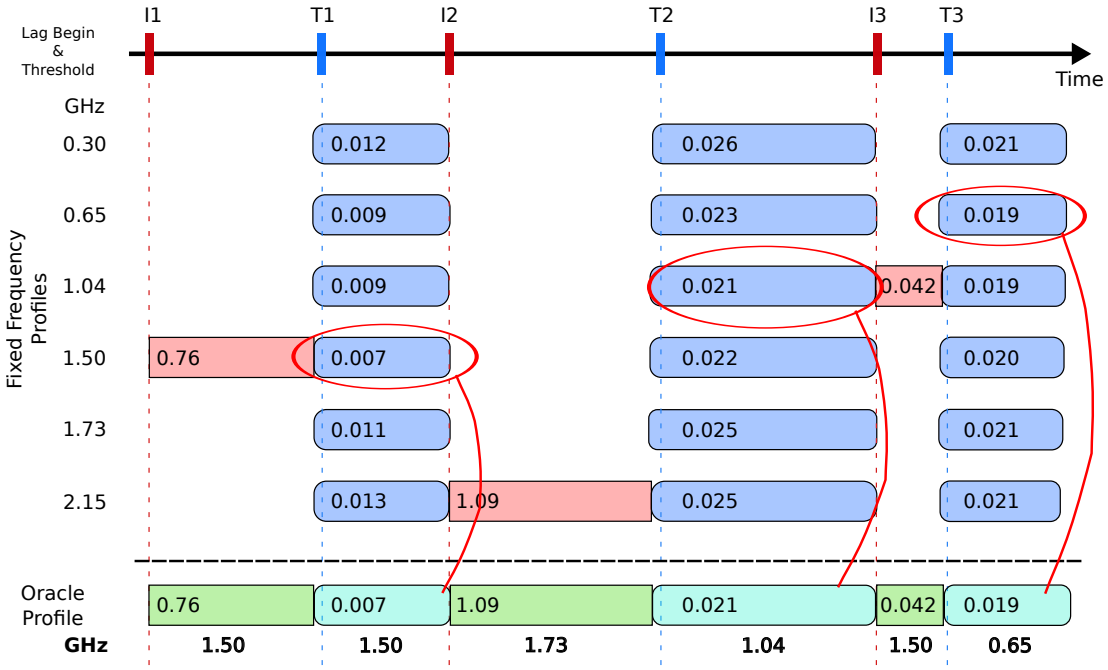


Figure 5.7: Pictogram showing how the *Oracle* algorithm selects idle frequencies for an example of three interactions. For each idle period (round blue box) frequency profiles are displayed showing idle duration (box size) and energy consumption in Joules (number in box). Frequencies 0.30 GHz - 2.15 GHz are available on the CPU for this example. The timeline on top marks interactions (I) and corresponding irritation thresholds (T). The boxes on the very bottom below the dashed line indicate which frequencies were chosen by the *Oracle*.

It is necessary to first select all lag frequencies before idle frequencies can be chosen. A lag's execution frequency influences the lag's duration, i.e. when it ends and the idle period starts. The interaction replay mechanism presented in Section 4.3.2 ensures that user input is issued at the exact same time for each benchmark execution. That means, between multiple workload executions, the same lags always start at the same time. Hence the following idle periods always end at the same time. The beginning of an idle period, however, is dependent on the lag duration which again depends on the selected lag frequency. Therefore, the *Oracle* needs to select a lag frequency first to fix a start point for the following idle period. This fixes the idle period duration. Then the *Oracle* can decide which idle frequency needs to be taken for lowest energy consumption.

Figure 5.7 shows how the *Oracle* selects idle frequencies for the same interaction example as in the previous section. For better visibility red square boxes showing lag statistics which were not selected by the *Oracle* are removed. Blue round boxes showing lag statistics for idle periods are added. Now that the length of the preceding lag is fixed for each idle period, idle begin and end are the same in each frequency profile. The *Oracle* now picks the frequency from the given selection which leads to the lowest energy consumption. As before energy does not scale linearly with frequency level. For the given example, selected idle frequencies are 1.50 GHz for the first idle period, 1.04 GHz for the second and 0.65 GHz for the third. This concludes the frequency selection process and the *Oracle* profile for the interaction example is finished.

5.4.3 Oracle Profile

Figure 5.8 shows a distribution of all interaction lag and idle frequencies selected for the *Oracle* frequency profile over the complete benchmark workload generated in Chapter 4. The medium frequencies are used most often due to their good balance between energy savings and lag duration and therefore user irritation. For idle periods the highest selection percentages of 22.6% and 24.4% have 0.96 GHz and 1.04 GHz. Frequencies lower than 0.96 GHz are rarely selected for neither lag nor idle. Higher frequencies are also rarely chosen for idle periods but more often for lags. The higher frequencies are used less for idle periods since here the focus lies on energy savings alone. Lags on the other hand need to consider irritation which pushes frequency distribution towards higher frequencies.

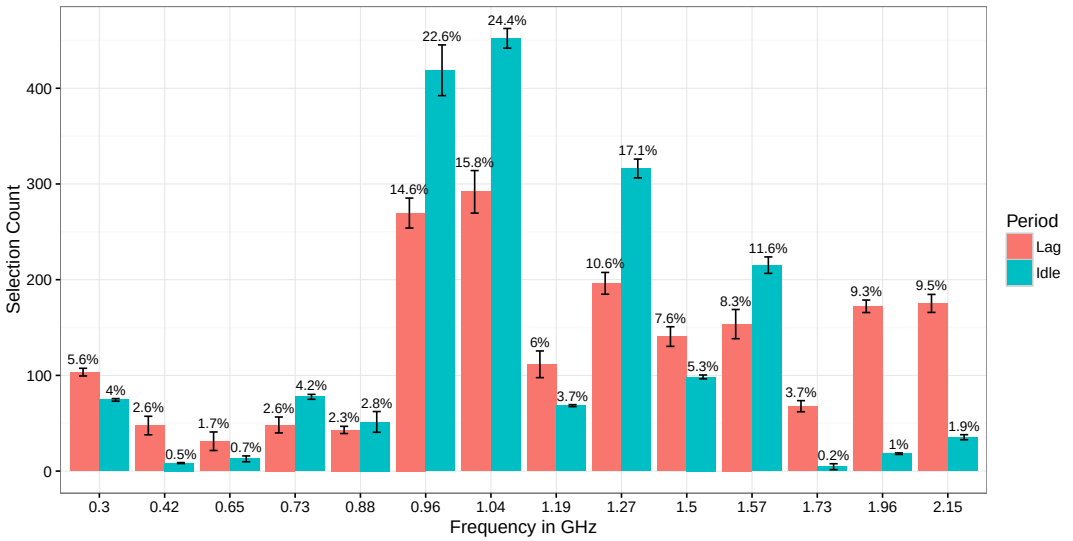


Figure 5.8: Distribution of frequencies selected for the *Oracle* profile. Left hand side bars represent the share for interaction lags, right hand side bars the share for idle periods. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

The *Oracle* generated a frequency profile which has minimal energy consumption whilst maintaining zero user irritation. Frequency selections for each period of the benchmark workload can now be used as a baseline to evaluate other DVFS approaches when executing the benchmark workload. Initially, this will be performed in the next section for the current standard DVFS techniques on *Linux* based mobile devices. Later, in Chapter 6 and Chapter 7 a new DVFS algorithm will be developed which avoids standard governor mistakes and achieves energy and irritation results close to the *Oracle*.

5.5 Experimental Setup

In the following sections experiments will be conducted where energy and irritation results of the *Oracle*'s workload frequency profile are evaluated against three standard frequency governors. Other user perception based DVFS studies on mobile platforms compare experimental results of their techniques against the same standard [134, 168]. The industry standard techniques for DVFS are represented by three Linux frequency governors used on modern mobile devices running *Android* operating systems. There is the *Ondemand* governor, the *Interactive* governor and the *Conservative* governor. *Ondemand* and *Conservative* are included in almost every modern Linux-based sys-

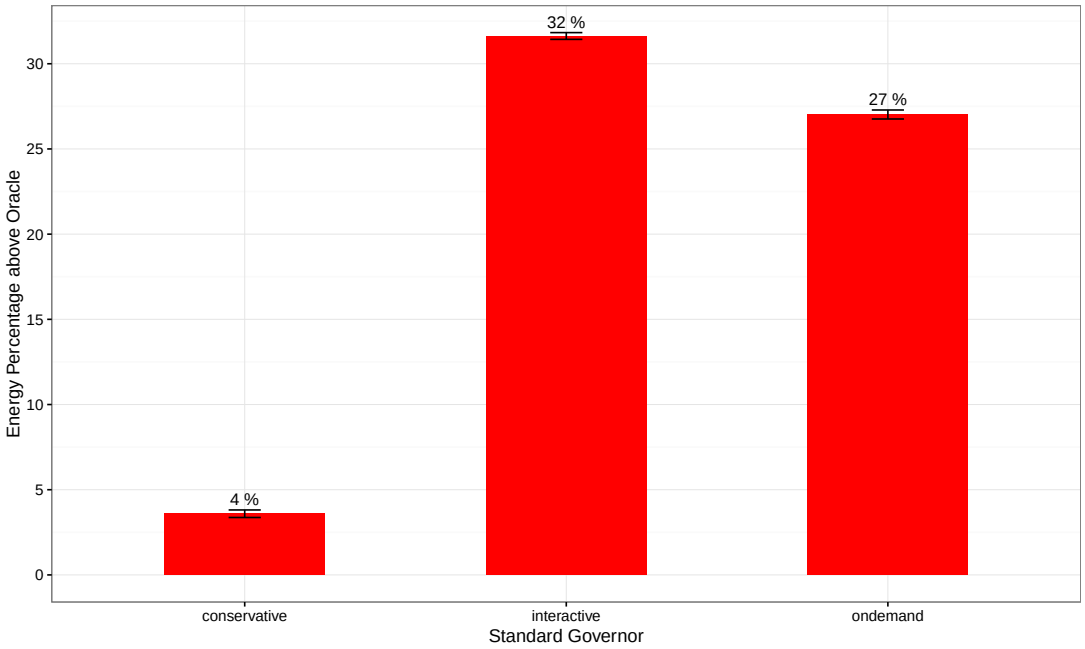
tem and *Interactive* is the standard governor for most *Android* mobile devices. All three base their DVFS strategies on the current load of each core. They ramp up the frequency as soon as the load raises above a fixed high-threshold and lower it again as soon as the load falls below a low-threshold. *Conservative* changes the load more smoothly than *Interactive* and *Ondemand* and stays longer in intermediate steps. *Interactive* has an additional feature where it reacts directly to incoming user input events and immediately ramps up the frequency while ignoring the load in those cases (see Section 2.6 for more details on their functionality).

The workload used for the frequency governor experiments as well as the hardware setup is the same as in Section 4.5. In contrast to previous experiments, the workload is now executed for each of the three frequency governors instead of fixed frequencies. Again, in order to minimise statistical error, each execution is repeated 5 times which leads to 15 executions of the entire workload. As before, the standard error is calculated across all 5 workload iterations.

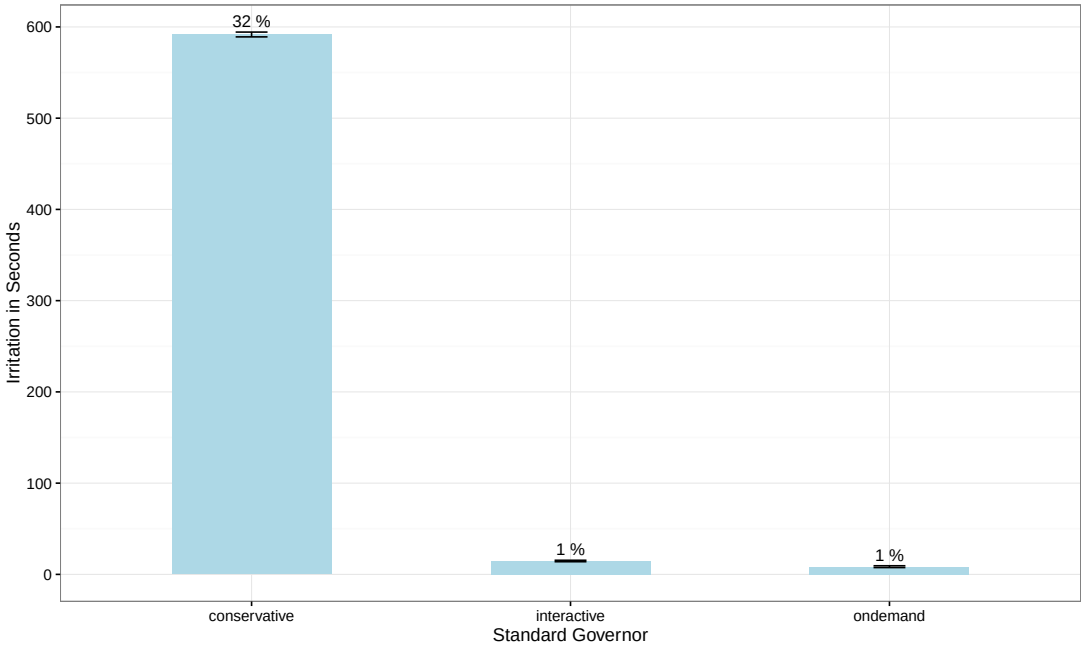
The frequency profiles generated by the governors and collected CPU load traces together with the power model from Section 5.3 are used to calculate the governors' energy consumption. Additionally, the user irritation metric from Section 5.2 is used to derive their corresponding irritation. The irritation thresholds are again set to 110% of what the fastest frequency could achieve for each lag. Resulting energy and irritation for each governor over the entire workload are then compared to each other as well as to energy and irritation for the *Oracle* performance profile generated in Section 5.4.3 and to all configurations with a fixed frequency from Section 4.6.

5.6 Experimental Results

First, total energy consumption and irritation of the entire workload execution are compared between the standard governors and the *Oracle*. This is followed by an analysis of where the governors select incorrect frequencies. Lastly, fixed frequency executions are compared to governors and *Oracle*.



(a) The energy consumption of each governor is displayed as percentage above the energy consumption used by the *Oracle*.



(b) User irritation is displayed in seconds. It is accumulated over all lag duration threshold violations of lags in the workload. The percentage value above each bar indicates for what fraction of total lag duration the irritation value accounts for.

Figure 5.9: User irritation and energy consumption of standard governors compared to *Oracle* results. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

5.6.1 Total Energy Consumption and Irritation

A summary of governor energy consumption and user irritation over the entire benchmark workload is shown in Figure 5.9. For all three governors, Figure 5.9a shows the percentage of total energy consumption above what the *Oracle* frequency profile can achieve. All three use more energy than the *Oracle* with the *Conservative* governor being closest with 4% and *Interactive* and *Ondemand* needing about a third more with 32% and 27% respectively.

Figure 5.9b shows the governors' accumulated user irritation over the course of the complete workload. The time value displayed on the irritation axis can be understood as a total time for which the user was irritated during the execution of the workload. The percentage above each bar shows which percentage of the total irritation lag time for each governor that was. With the *Oracle* frequency profile not being irritating at all, the *Conservative* governor is furthest away with a total irritation of 593 seconds. This is 32% of the total interaction lag time of the workload. It can be interpreted as the end user being irritated 32% of the time while waiting for the system to finish processing his interaction. *Interactive* and *Ondemand* both have a total irritation of roughly 10 seconds which accounts for 1% of the total interaction lag time.

In conclusion it can be said that all three governors have room for improvement. Even though *Interactive* and *Ondemand* are doing quite well when it comes to providing good QOE they require about a third more energy to execute the workload. *Conservative* needs only a little more energy than the *Oracle* profile but is irritating to the user for about a third of the time.

5.6.2 Governor Frequency Compared to *Oracle*

Figure 5.10 shows a break down of where frequency selections by the single governors differ from the *Oracle*. This is shown by using average frequencies over all user perceived workload periods, i.e. all irritation lag and idle periods of the full workload. The governor's average frequency for each period is compared to the *Oracle*'s equivalent and it is counted how often it is below, above or the same as the *Oracle*'s choice. A governor's average frequency over a workload period is indicated as follows:

All frequencies selected by the governor over the course of a single workload period, i.e. a single interaction lag or idle period, are weighted by their duration. The sum of

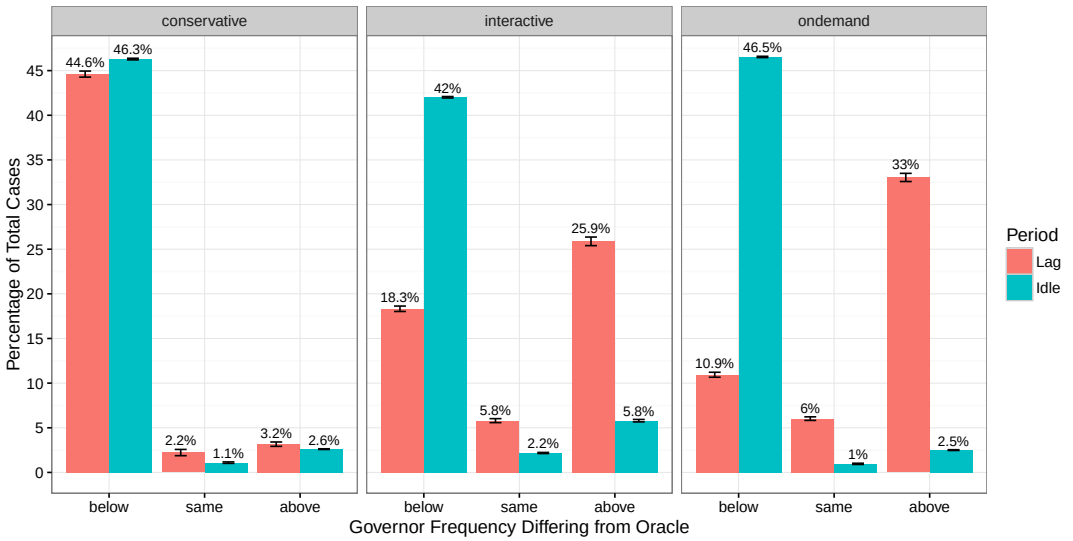


Figure 5.10: Difference of average governor frequency compared to *Oracle* frequency for each perceived workload period. Left hand side bars represent the share for interaction lags, right hand side bars the share for idle periods.

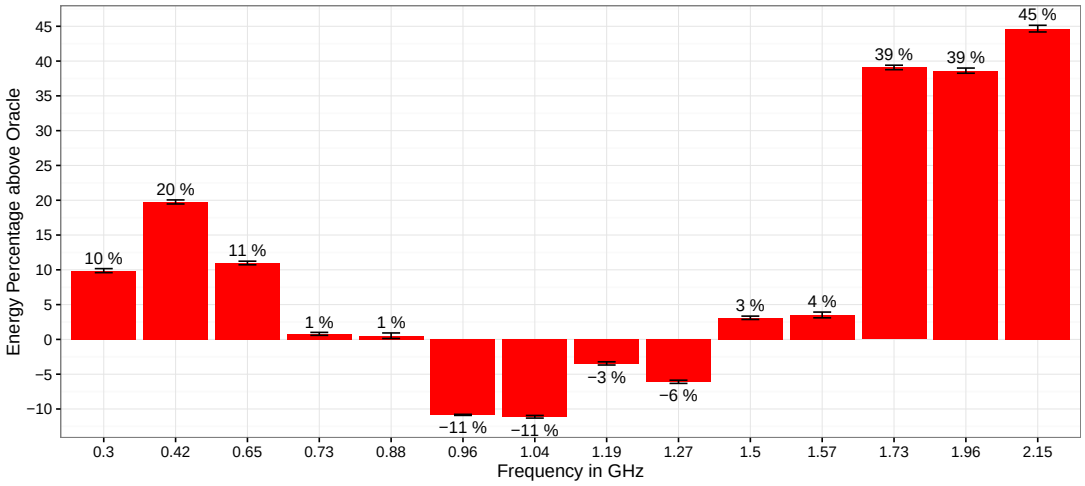
these weighted frequencies is divided by the total duration of the workload period. The result is rounded to the nearest available frequency on the processor's scale.

From the bar pairs shown in the figure, the left bar represents the share for interaction lags out of all workload periods while the right bar represents the idle period share. It can be seen that in about 90% of the cases the *Conservative* governor chooses a frequency that lies below the frequency chosen by the *Oracle*. 44.6% of those belong to interaction lags which account for about the same energy consumption as idle periods in the total workload and for all of the user irritation. This explains the results from Figure 5.9 where the *Conservative* governor is much more irritating than the *Oracle*. With the average frequency being below the *Oracle*'s the interaction lags are longer and therefore user irritation grows.

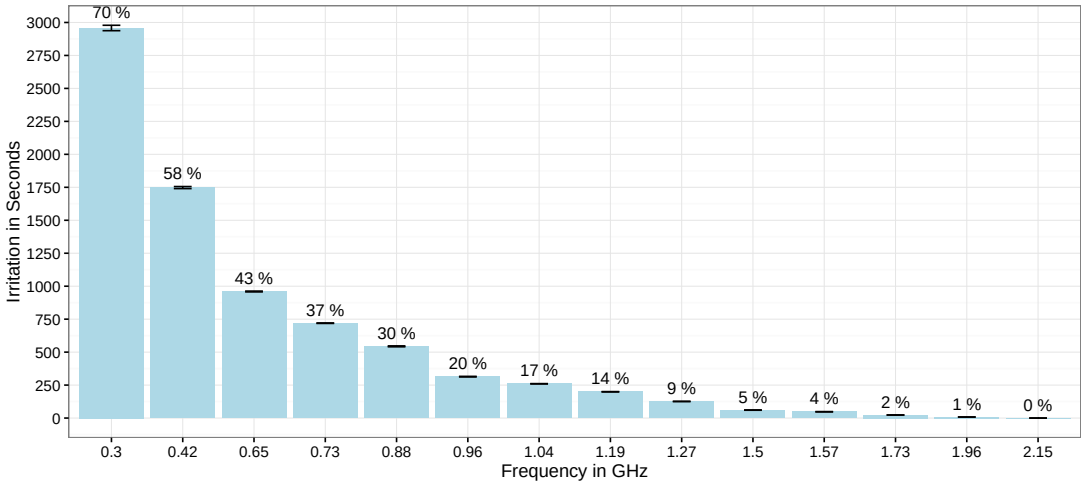
Interactive and *Ondemand* show similar results. Like *Conservative* they stay below the *Oracle*'s frequency during idle periods for 42% to 46.5% of the time. However, in contrast to *Conservative* they exceed the *Oracle*'s frequency during lags in 25.9% and 33% of the cases. This is the reason why the *Oracle* profile is doing better in terms of energy efficiency compared to those two governors. The fact that the idle period is below the *Oracle* profile so often and still governors need more energy is bound to the amount of work that needs to be done. All three governors scale frequency with the CPU load which is close to zero for most of the idle time. For occasional load spikes at idle time due to background processes, governors raise the frequency and lower them

again shortly after. Most of the time, however, their selected frequency is the lowest possible. This causes the average frequency, which is weighted by duration, to be lower than what the *Oracle* picked. The *Oracle* having perfect knowledge of which frequency works best for the entire period, anticipates load spikes and selects a higher one.

5.6.2.1 Fixed Frequency Compared to the *Oracle*



(a) The energy consumption of each frequency is displayed as percentage above or below the energy consumption used by the *Oracle*.



(b) User irritation is displayed in seconds. It is accumulated over all lag duration threshold violations of lags in the workload. The percentage value above each bar indicates for what fraction of total lag duration the irritation value accounts for.

Figure 5.11: User irritation and energy consumption of fixed frequency configurations compared to *Oracle* results. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

In addition to the governors, the *Oracle* profile is now compared to performance configurations where the core was fixed to a certain frequency. Figure 5.11 shows the corresponding results for energy and irritation. Figure 5.11a shows energy consumption for all available core frequencies compared to the *Oracle* profile. Always running on the fastest frequency needs 45% more energy than the *Oracle* selection. The 8 medium frequencies are close to the *Oracle*'s energy consumption while 0.96 GHz to 1.27 GHz manage to use less.

Figure 5.11b show user irritation over the course of the workload. Again, the percentage above each bar shows which percentage of the total irritation lag time the accumulated irritation accounts for. It degrades the faster the frequency becomes while the fastest frequency is per definition not irritating (see irritation threshold specifications in Section 5.5). However, the fastest frequency is the only frequency to achieve an irritation of zero. The *Oracle*'s frequency settings achieve a user experience that is indistinguishable from always running at the highest frequency while using 45% less energy.

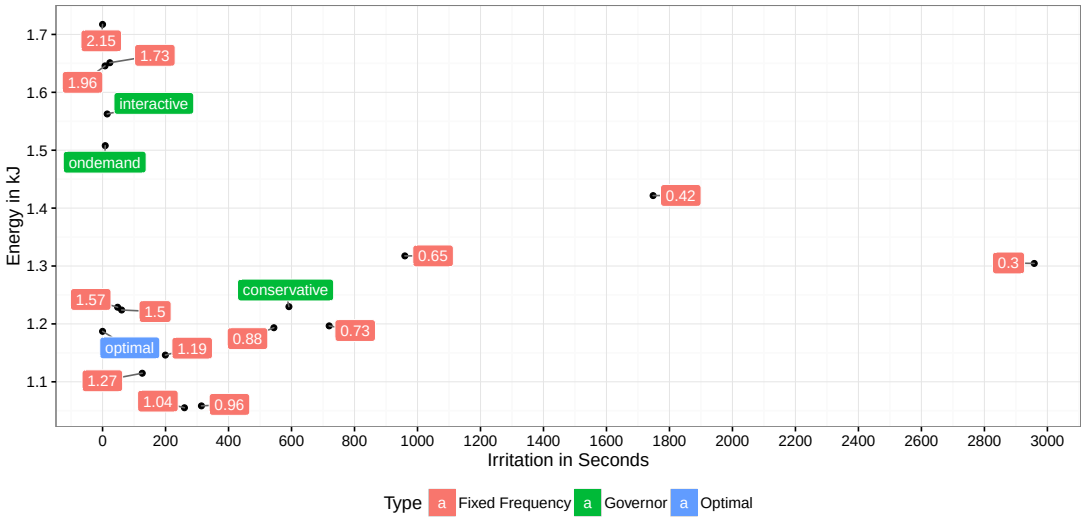


Figure 5.12: Scatterplot of energy and irritation metric for the workload with frequency governors and fixed frequencies. *Oracle* profile and the fastest frequency both have an irritation of zero per definition.

Figure 5.12 shows a comparison of all fixed frequency profiles, all governors and the *Oracle* by plotting energy over user irritation for the whole workload. It shows that some performance profiles reach the *Oracle* in terms of user irritation and some come close or even exceed it in terms of energy consumption, but none surpasses it in both. It is also noteworthy that some fixed frequencies, namely 1.57 and 1.50 GHz, are

performing nearly as well or better than the governors. This is likely due to the nature of the analysed workload. A longer workload with more interaction examples and different mobile platforms can help to investigate this further.

5.7 Conclusion

In this chapter the energy saving potential of current *Android* frequency governors was evaluated while considering user perception. In order to do that the benchmark workload and interaction lag marking method from Chapter 4 was used. To quantify QOE for a workload execution, a metric was introduced which evaluates interaction lag data generated by setting maximum deadlines for each interaction. For the benchmark workload an *Oracle* profile was created that would use the least possible energy whilst still being able to meet interaction deadlines without irritating the user. When comparing the governor frequency profiles with the *Oracle*'s, results showed that *Interactive* and *Ondemand* leave room for more energy savings with up to 32% while the *Conservative* governor needs on average 4% more energy than the *Oracle*. *Conservative* shows, however, a significantly higher user irritation. On average the user is irritated for 32% of the time while waiting for the system to finish processing his interaction. *Interactive* and *Ondemand* need on average 32% and 27% more energy but are a lot closer to the *Oracle* in terms of irritation (on average less than 1%).

This chapter could successfully demonstrate the usefulness of the method developed in Chapter 4 and how it can be applied to measure QOE of a DVFS approach. In the following chapters a DVFS technique will be developed which considers interaction lag durations as perceived by the user. It avoids frequency selection mistakes made by the standard governors and is able to match the *Oracle*'s results more closely.

Chapter 6

Runtime Interaction Lag Detection

6.1 Introduction

In the previous two chapters, novel methods and tools were introduced to accomplish the goal of improving energy efficiency of DVFS techniques for mobile workloads. An interactive mobile workload was generated from input recordings of real users. It was used in conjunction with a methodology to mark interaction lags and a metric to quantify user irritation. Executing this tool chain allows benchmarking a DVFS technique in terms of QOE and energy efficiency. The feasibility of this approach was demonstrated in the previous chapter: There, current standard frequency governors were evaluated by comparing their selected frequency profiles against an *Oracle* baseline profile with minimal energy consumption and zero user irritation. Results showed that the governors leave room for improving energy efficiency without applying user perceptible performance degradation.

In this and the following chapter, an improved DVFS technique is developed. By considering findings of the work so far, the new technique is designed to make frequency decisions based on user perceived workload periods. Therefore, a crucial requirement to implement such a technique is runtime knowledge of when the system executes a lag and when it is idle as seen from the user's point of view. This is because those periods have different requirements for being handled by a QOE aware governor. Such a governor needs to pick a high enough frequency during interaction lags to keep irritation low yet without wasting energy. During idle periods it can ignore irritation and focus on using the most energy efficient frequency. This chapter will describe in

detail how the developed governor is able to detect the boundaries between interaction lag and idle periods as seen by the user. The QOE aware governor developed in the next chapter will then use this information to avoid mistakes made by current standard approaches and match the *Oracle* baseline more closely.

Recent studies exist that are trying to capture user interaction periods by instrumenting the layout tree of *Android* applications [9] or tracking the end of all tasks or threads triggered by an interaction [138, 139, 152, 168] (see Chapter 3 for details). They show, however, limitations in the type of interactions they can handle or require extensive OS framework instrumentations. More importantly none of them uses a systematic method to evaluate their technique against actual screen output seen by the user. The heuristic developed in this chapter is tested for interactions across multiple applications recorded from real users and evaluated against the video frame markup method introduced in Chapter 4.

6.1.1 Runtime Detection of Lag and Idle Periods

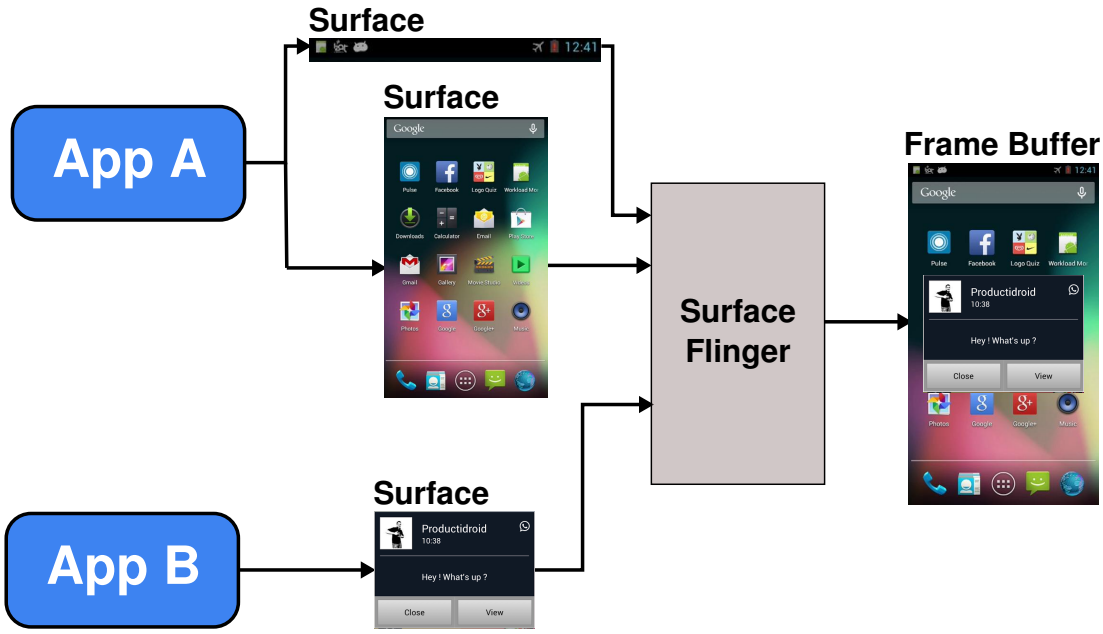


Figure 6.1: *Android SurfaceFlinger* component combines drawable surfaces of active applications to the final frame buffer image presented on the screen.

Detecting the start of an interaction lag can simply be performed by tracking user input events. The real challenge lies in finding the start of an idle period, i.e. the end of a lag. Since this boundary is defined by the user's visual perception of the workload,

the governor needs to have low overhead access to information on the screen output. For the offline approach in Chapter 4, a video was recorded of what the screen was showing and its frames were analysed to identify interaction intervals. This approach, however, is too complex and time consuming to be used at runtime. Instead, screen refresh executions of the *Android* framework's *SurfaceFlinger* component, CPU load and input events are considered to get an idea of the user's perspective.

SurfaceFlinger is a framework component responsible for producing the final screen output image. Its functionality is displayed in Figure 6.1. This display stack component combines drawable surfaces of all currently active applications (see Section 2.5.1 for details). After an application finishes rendering screen content into a surface graphic buffer, it notifies *SurfaceFlinger* about the new content. *SurfaceFlinger* then executes a screen refresh for all currently available surface buffers. Screen refreshes, therefore, correlate well with frame changes in a workload video as seen by the user. The video markup method introduced in Chapter 4 identified the first video frame in a series of still standing images as the ending of an interaction lag. The Lag End Detection Heuristic (LDH) developed in this chapter finds the corresponding screen refresh by monitoring system statistics at lag execution time.

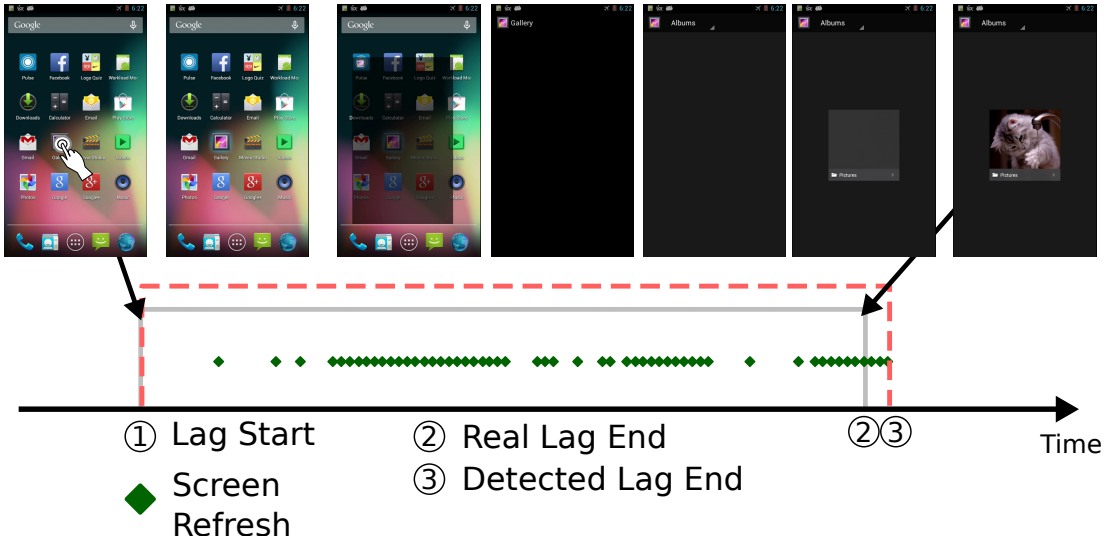


Figure 6.2: Interaction lag from both the user's and system's perspective. The user's perspective is represented by screenshots of the device's output and the system's perspective shows *SurfaceFlinger* screen refreshes (green diamonds). The correlation between those perspectives is demonstrated using two lines. Dashed red shows lag dimensions detected by the heuristic, solid grey shows dimensions as seen by the user, i.e. as detected by the lag marker method from Chapter 4.

Figure 6.2 demonstrates user and system perspective for an example interaction. It shows *SurfaceFlinger* screen refreshes, visual presentation to the user and lag endings as seen from both sides. In the upper part of the figure a chain of screenshots shows how the user perceives the executed interaction of opening the Gallery application. The solid grey line above the timeline marks the lag period as seen from the user's point of view and the dashed red line indicates the lag period as detected by the runtime LDH. Each green diamond marks a screen refresh execution of *SurfaceFlinger*. In the example, the user perceived lag ending comes first and then the screen refreshes stop which the runtime heuristic reports as a lag end. This is the most common case for interactions in the workload generated in this study. The causes for the offset between detected ending and last screen refresh will be explained in Section 6.2.

The LDH accuracy evaluation presented at the end of this chapter shows that reported lag endings can sufficiently capture the user's perspective. The last *SurfaceFlinger* screen refresh of a lag which is reported as a lag ending by the heuristic diverges from the lag ending as seen by the user with an average error of 11.7%. It will be shown in Chapter 7 that this accuracy is sufficient for improved energy efficiency and a reduction in user irritation.

6.1.2 Contributions

The contributions of this chapter are:

1. An analysis of runtime statistics at interaction lag times and their correlation to visual output as seen by the user and,
2. a runtime heuristic to determine the ending of user perceived interaction lags.

6.1.3 Overview

Section 6.2 will demonstrate how system statistics and visual representation correlate using multiple examples. Implementation details of the LDH and how it is used to determine the lag ending during execution will be described in Section 6.3. The experimental setup and metrics for an evaluation of the heuristic's detection accuracy are presented in Section 6.4 and corresponding results are discussed in Section 6.5. Section 6.6 will then summarise and conclude the chapter.

6.2 Correlation between System and User Perspective

The improved governor's LDH must be able to detect an ending for various kinds of interaction lags. Some with a duration of only a few hundred milliseconds others with durations of up to several seconds. It must be able to find an ending for different CPU frequencies which show different execution statistics for the same lag. Furthermore, the heuristic must be able to determine if system activity is still part of the current lag or background activity of the following idle period. Also discrepancies between the user perceived screen output and *SurfaceFlinger* screen refresh events need to be considered. This section will show examples of different interaction lag scenarios including user output and underlying system statistics to give an idea of how the statistics used by the LDH look like.

6.2.1 Long and Short Lags

Figure 6.3 shows graphs of the system statistics used by the LDH for a longer (1.5 seconds) and a shorter (0.3 second) lag. Above each graph is a series of screenshots showing the screen output at the time of the execution. The y-axis shows a percentage scale for CPU load. The busy time of a single CPU is measured over 10 millisecond intervals and the busy to idle time ratio is displayed in the graph as a thin red line. Along the 0.8 level of the y-axis blue dots represent input events as processed by the *Linux* subsystem driver (see Section 4.3.2). Screen refresh events issued by *SurfaceFlinger* are displayed as green diamonds along the 0.5 level of the y-axis. A thick grey line marks the boundaries of the lag as seen by the user, i.e. as detected by the lag marker methodology described in Section 4.3.4.

The 1.5 second lag shows how a tap on the home screen shortcut of the Gallery application is registered and how the application starts. The first cluster of input event dots represents where the finger touches the screen and the second one shows where it is removed[1]. A series of screen refresh clusters follows representing the screen changes during application startup. First, the shortcut glows in blue, then a black background fades in from the centre, the headline is changed from "Gallery" to "Albums", a grey square appears as a place holder for the first picture album and lastly the header image of the album fades in. After that, the lag is over and screen refreshes stop.

[1] Input events are so close in succession that they might appear as one dot in the graphic.

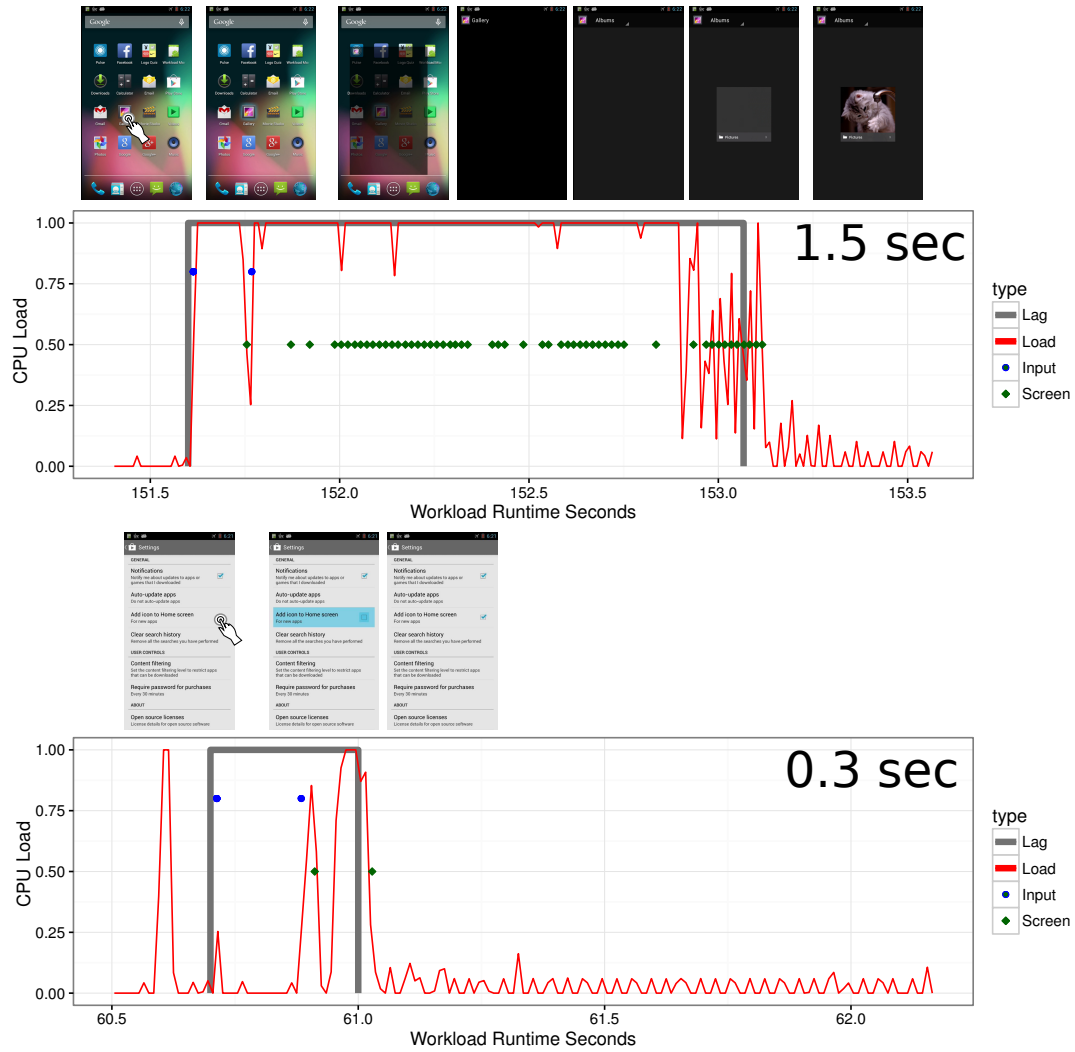


Figure 6.3: Visual output and system statistics of a long interaction lag of 1.5 seconds compared to a short lag of 300 milliseconds. User perceived lag dimensions are shown as a thick grey lines, CPU load as a thin red lines. Input events are displayed as blue dots and screen refreshes as green diamonds.

The CPU load spikes up as soon as the screen is touched. It has a small dip where the finger leaves the screen. Afterwards, it stays up around 100% until the application is loaded. Around 153 seconds, where the header image of the album is fading in, load goes down and only spikes shortly around 50% for each screen refresh. When the lag ends, load lessens further and stays low around 10%. The CPU for the graphs in this figure is fixed to a frequency of 1.04 GHz which is from the medium range of all available ones on this core. In the beginning most of the components needed for the application are loaded, which is finished around 153 seconds. Afterwards, the CPU is busy drawing each screen refresh until the screen changes finally settle and the lag is over. The following CPU activity is due to background tasks of the application and the

OS.

The short 0.3 second lag displays how an option in a settings menu is ticked to activate it. The initial finger-down-and-up input is followed by two screen refreshes: a blue highlight appears and disappears again and a tick mark is set. CPU load spikes shortly for each screen refresh. As for the examples, long lags usually contain multiple clusters or bursts of *SurfaceFlinger* screen refreshes while short lags can have only one or two single refreshes. The LDH must be able to handle both cases. The CPU load can alternate quickly or stay high for larger intervals. It usually declines after the lag ends. Finding the last burst or the last singular screen refresh in a lag and a declining load are therefore a good indicator for a lag ending. A burst ending and declining load can, however, also happen within a lag, especially for higher frequencies. It is also possible that an animation during idle time directly following the lag causes more screen refreshes which are not part of the actual lag. These cases also need to be covered by the heuristic.

The two examples show again that the lag ending detected by the lag marker method from Chapter 4 has a slight offset to the last screen refresh. In the long lag, it is caused by the frame comparison technique used by the lag marker method. In this method, frames are determined to be equal by comparing their pixels. During pixel comparison, a small colour difference is allowed to compensate for noise and video artefacts. Minor pixel changes which are actually part of the lag can be masked by the noise which causes the lag marker method to report a premature ending. This can happen during screen transitions where a fading animation is used. The user, however, is not likely to detect those differences himself.

The offset in the short lag is caused by the granularity with which the lag marker method works. It considers lag endings only for frame boundaries which are 33.33 ms apart for a video of 30 fps. The actual system screen refresh can therefore diverge from the reported ending for up to that much. Again, this difference is not noticeable by the end user.

6.2.2 Different CPU Frequencies

Figure 6.4 shows how different CPU frequencies affect the outcome of a lag execution. Three frequencies are displayed: the lowest possible on top with 0.3 GHz and a lag

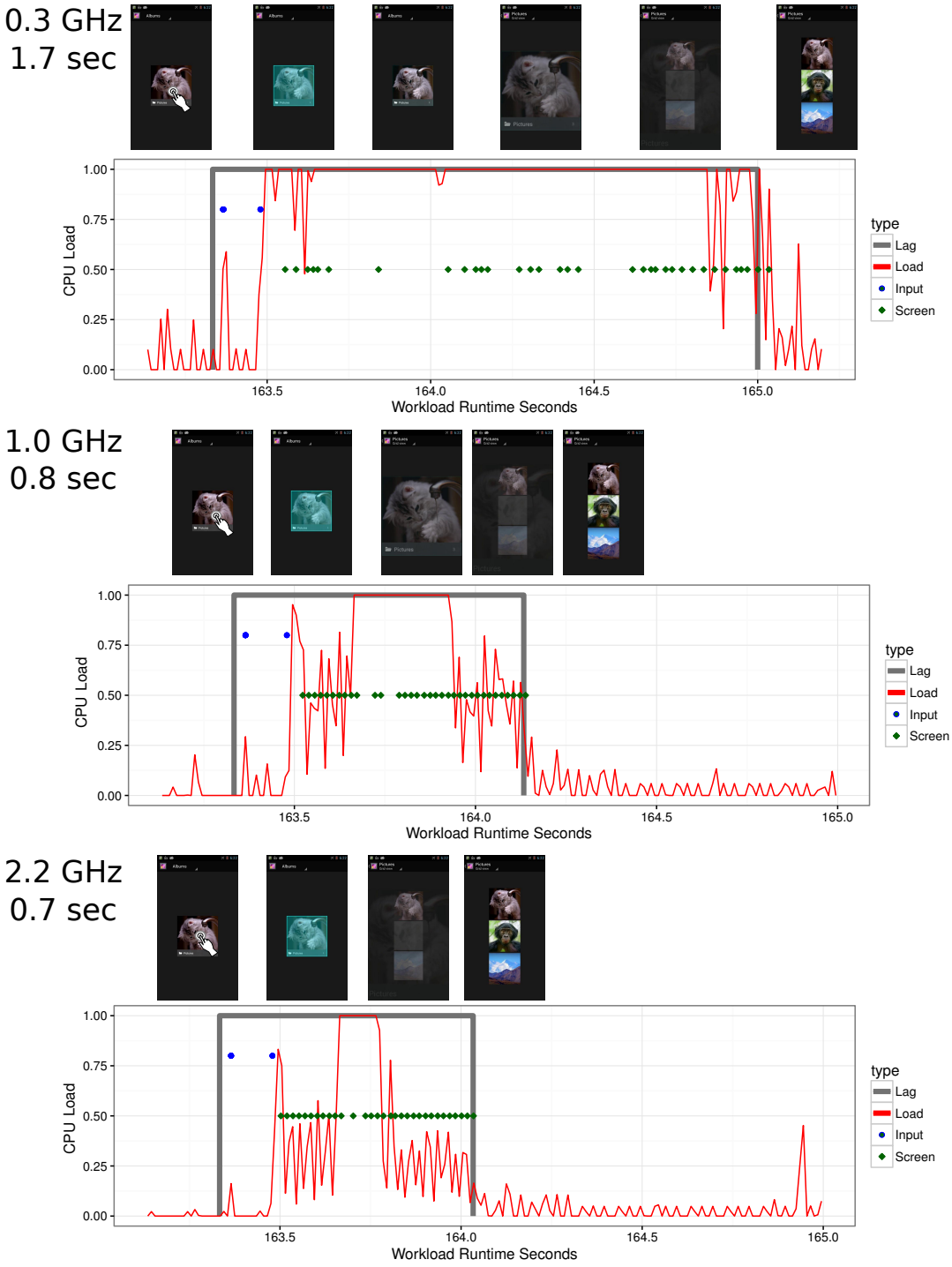


Figure 6.4: Visual output and system statistics of the same interaction lag for three different CPU frequencies (0.3 GHz, 1.0 GHz and 2.2 GHz). User perceived lag dimensions are shown as a thick grey line, CPU load as a thin red line. Input events are displayed as blue dots and screen refreshes as green diamonds.

runtime of 1.7 seconds, a medium frequency of 1.0 GHz in the middle with a lag runtime of 0.8 seconds and the fastest frequency on the bottom with 2.2 GHz and a lag runtime of 0.7 seconds. The lag itself corresponds to an interaction where the user taps on an album in the Gallery application. First, the album is highlighted in blue and then fades into showing its picture content. While input events stay the same, load and screen refresh statistics differ between the three frequencies. On the fastest frequency three screen refresh clusters and one high load spike can be seen. The three clusters belong to the fade-in animation of the album highlight, the disappearance of the highlight and the fade between album and picture content. Components of the album content are loaded as soon as the highlighting process is finished. The lower the frequency, the longer the component loading process takes and the longer the lag duration is stretched.

After an application finishes drawing its screen content to a surface, it notifies *SurfaceFlinger*. *SurfaceFlinger* then refreshes the screen content by compositing visible surfaces of all currently running applications depending on where they are located. A synchronisation framework ensures that screen refreshes happen at the frame rate of the display device (30 fps on the device used in this study). If the screen content does not change between two frames, *SurfaceFlinger* does not issue a refresh and the display device can simply draw the same graphic buffer content as before. On the lowest frequency the CPU cannot always handle surface rendering, final surface composition and potential additional background tasks within the time limit of a single frame (33.33 ms for 30 fps). If the CPU is too busy, screen refreshes might be dropped and the screen output's frame rate "stutters" and appears unsmooth. Therefore, execution of the same lag for lower frequencies usually has more clusters of screen refresh intervals which are spread out longer than execution for higher ones. The LDH must be able to track different amounts of screen refresh clusters in a single lag depending on current frequency settings.

6.2.3 Idle Time Activity

Figure 6.5 shows an example of system activity between lags, i.e. during the idle period. The two lags displayed here show the menu of the Logo Quiz application. Play is tapped in the first lag which leads to a sub menu displaying available levels. The first level is selected in the second lag which brings the device to a screen showing

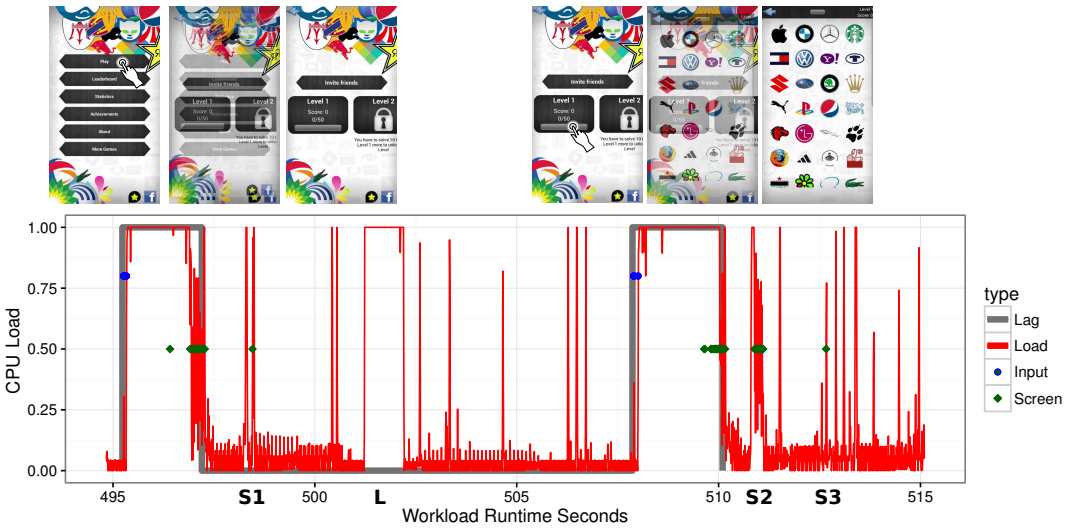


Figure 6.5: Visual output and system statistics for idle time period following two lags. User perceived lag dimensions are shown as a thick grey line, CPU load as a thin red line. Input events are displayed as blue dots and screen refreshes as green diamonds.

available logos to guess in this level. Within the idle period between the two lags are multiple short load spikes and a longer one around 502 seconds marked with **L**. This long load spike is due to a background *Google Login Service* trying to synchronize content for an application using a *Google* account. Around 498, 511 and 513 seconds marked with **S1** to **S3** are screen refresh events with corresponding load spikes during idle time. Among them, only the screen refresh burst at **S2** leads to screen changes actually visible to the user (more details on imperceptible screen refreshes in the next section). Here, a scroll bar fades out after the logo overview was loaded. The LDH must be able to decide which screen refresh and load spikes belong to a lag and which are inside an idle period.

6.2.4 Screen Refresh Inaccuracies

Although *SurfaceFlinger* screen refreshes are a good indicator of when the screen changes and what the user sees, this statistic is not ideal. There are discrepancies between screen refreshes and actually visible changes on the device's display. It can happen, that screen refreshes are scheduled in *SurfaceFlinger* which result in screen changes that are so minor, that the human eye cannot detect them or only with great difficulty. For the mobile user and the video frame lag detection method from Chapter 4 this looks like the screen is actually standing still. It is also possible that an application

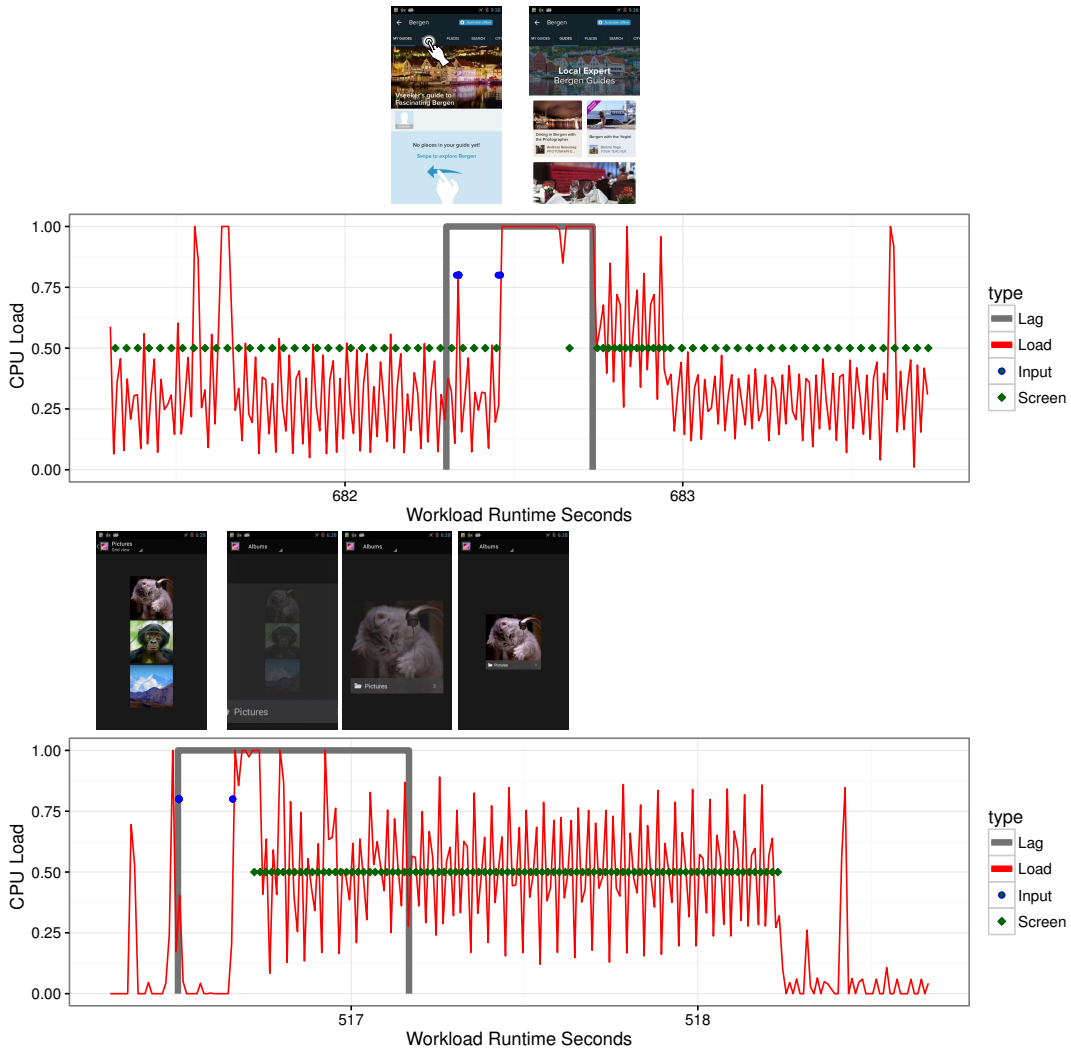


Figure 6.6: Visual output and system statistics showing discrepancies between screen output visible to the user and *SurfaceFlinger* screen refresh events. *SurfaceFlinger* refreshes the screen without perceptible screen changes. User perceived lag dimensions are shown as a thick grey line, CPU load as a thin red line. Input events are displayed as blue dots and screen refreshes as green diamonds.

signals to *SurfaceFlinger* that a new graphic buffer is ready but it does not actually differ from the previous one. A screen refresh follows but the screen content does not actually change. Figure 6.6 gives two examples for this behaviour.

The upper graph shows an interaction with the Stay tourism application. This application provides city guides or information on various tourism goals around the world. The interaction is a tap on the top menu where the application switches from showing favourite guides to a list of available ones. The interaction is executed on a medium CPU frequency and is about a third of a second long. During the actual screen update the content below the top menu rolls to the left when the next menu entry is selected.

It happens so fast during execution that the animation is barely visible. It is shown as a single screen refresh within the high CPU load region towards the end of the lag. However, the idle periods surrounding the lag show a high rate of repeated screen refreshes even though no differences between successive frames are perceptible.

The lower graph again shows an interaction with the Gallery application. Here, the picture content of an album is displayed and the back button is tapped. The picture content fades out and the available album overview fades in. The video frame lag detection method detects a stop in visible screen changes after 0.7 seconds, i.e. the user perceived end of the lag. However, the *SurfaceFlinger* screen refreshes are being scheduled for 1.7 seconds which is nearly three times as long.

Three different people were asked to have a look at the actual screen output for those lags and could not see any screen changes for the times in question. Also, the video frame detection method could not find significant pixel differences besides noise. This kind of behaviour can result from badly written code which requests screen refreshes from the *Android* framework without ensuring it is necessary. Another possibility are pixel changes so minor, that the human eye and the video frame comparison cannot detect them. The following section will present the LDH algorithm and explain how it handles the cases presented above.

6.3 Lag End Detection Heuristic

The runtime heuristic to detect lag endings works with the system statistics presented in the previous section. It monitors system events and reports the ending of the lag currently being executed as seen from a user's point of view. In so doing, it needs to overcome the following challenges:

1. An ending needs to be detected by observing screen refresh clusters and CPU load spikes for lags with various durations.
2. The load profile and screen refresh clusters of the same lag look different for different frequencies.
3. System activity such as load spikes or screen refreshes can appear within idle periods (e.g. scroll bar fade outs right after the lag ended, a cursor blinks, the clock changes or a CPU-heavy background task is active).
4. Visible screen changes and system screen refreshes do not always align.

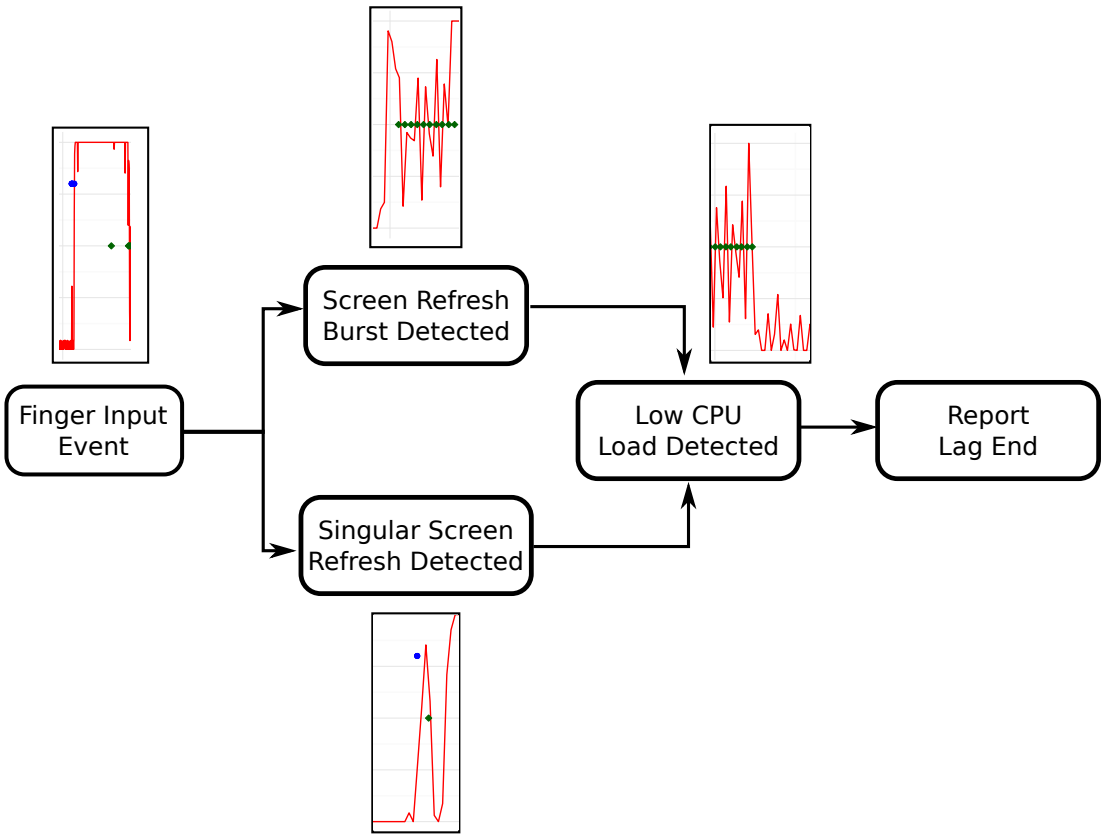


Figure 6.7: LDH workflow: After the interactions initial input events, system statistics are monitored to find at least one screen refresh burst (upper branch) or a singular screen refresh (lower branch). When one of the branches was taken, LDH waits for the CPU load to drop. As soon as the load drops and stays low the lag end is reported.

A high level workflow of the LDH is shown in Figure 6.7. Detection starts after a finger input event was observed. This is usually a finger down event followed by a finger up event. However, the heuristic is also able to handle cases where the finger remains on the device longer than for a quick tap, i.e. swipes or long touch events. The heuristic aims to capture user perceived lag endings, which are tied to screen refreshes. Therefore, after an input was detected, it monitors *SurfaceFlinger* activity and looks for bursts of screen refresh events (upper branch in the figure). Such a burst is a series of screen refreshes following each other in regular time intervals. The intervals between refreshes of a single burst depend on the currently running application and can range between 16 ms and 500 ms. Specific expected properties of a burst such as refresh interval length and minimum count of successive refreshes to be considered a burst, can be configured. As soon as the ending of a burst was detected, the heuristic waits for the CPU load to settle below a background activity threshold. A burst followed by low load is a strong indicator for a lag ending. If the load stays up, usually

another burst follows because the lag is still active. Short lags often only have one or two screen refresh events which do not add up to a burst. Therefore, the heuristic also starts looking for low CPU load if no burst was detected yet, but at least one singular screen refresh appeared (lower branch in the figure).

Figure 6.8 shows a pseudo code algorithm of the LDH. The displayed function is called in regular intervals while a lag is active. When the lag end was detected, it returns the time of the last screen refresh in the lag as result (line 13). The function starts checking for a lag end as soon as either the user's input finger left the screen or touches the screen for longer than a configurable amount (300 ms for experiments in this study). The upper branch in Figure 6.7 is represented in pseudo code by the condition in line 6: When at least one burst of screen refreshes was observed and the most recent burst already ended, the first condition for a lag end is fulfilled. The lower branch is represented by the second condition in the same if clause in lines 7 and 8: If no screen refresh burst has been observed yet but a minimum configurable amount of singular screen refreshes was seen (one, for experiments in this study), a lag end is potentially imminent. In lines 10 and 11 the final condition for a lag end is checked: When the load since the last observed screen refresh is lower than a configurable threshold (5% for the experiments in this study) and a configurable amount of time has passed since then (30 ms for experiments in this study), a lag ending was found.

```
1   function detectLagEndTime() {
2     // SR: Screen Refresh
3     if(fingerLeftScreen |
4        fingerOnScreenDuration > HOLD_EVENT_THRESHOLD) {
5
6       if((SRBurstCount > 0 & lastSRBurstEnded) |
7          (SRBurstCount <= 0 &
8            SRCountAfterFingerDown > SHORT_LAG_MIN_SR_COUNT)) {
9
10        if(loadSinceLastSR <= LOW_LOAD_THRESHOLD &
11           timeSinceLastSR >= LAG_END_THRESHOLD) {
12
13          return timeOfLastSR;
14        }
15      }
16      return null;
17    }
18  }
```

Figure 6.8: LDH end detection algorithm. It is invoked regularly when a lag is active and reports the time of the last observed screen refresh when a lag end was detected.

When the conditions for a lag ending are met, the heuristic reports it to the governor. Waiting for the load to remain low before reporting an end usually leads to a small offset between the last observed screen refresh and the reported end. This is necessary to ensure that load or screen refreshes are not picking up again right after the last seen burst ending because the lag is still continuing. To counteract this inaccuracy the total lag duration passed on to the governor does not include the mentioned offset. It ranges only from input until the last observed screen refresh. This helps to align more closely with the user's point of view. To reduce runtime overhead, the heuristic scans execution statistic in regular sample intervals (10 ms by default). The offset caused by this sampling distance is, however, negligible when considering that a user perceptible frame update takes 33.33 ms at 30 fps.

The heuristic's algorithm in this form is able to tackle most of the challenges listed above. The only problematic lags are the ones where screen refreshes and actually visible changes do not align. Here, the heuristic can have a high detection error compared to the lag ending as the user sees it. However, these cases are not frequent in the workloads collected for this study. Additionally, the resulting inaccuracy is low enough to not affect the overall goal of energy efficiency and high QOE. A detailed evaluation of LDH detection accuracy will be presented in the next section.

6.4 Experimental Setup

The experiments presented in this chapter serve to evaluate the detection accuracy of the Lag End Detection Heuristic presented above. Detection results of lag periods are compared to lag duration measurements performed by the method presented in Chapter 4. Inaccuracies between the results of runtime heuristic and video frame method are reported using a mean absolute percentage error metric. This metric will be described in detail in Section 6.4.2.

The benchmark workload executed for the experiments is the one generated in Chapter 4. It is composed of 16 datasets from different users with a length of about 10 to 15 minutes each. It has a total length of 190 minutes with 1935 user input events. Just like for the experiments in Chapter 4 and Chapter 5, the leading and trailing interactions of each dataset were omitted. They are used to activate and deactivate workload recording and are therefore not part of the actual workload. Also, some interactions

within the workload were omitted since the recording method presented in Chapter 4 could not handle them (see detailed workload description in Section 4.4). This leaves 1852 interactions for the analysis.

To comprehensively measure lag durations as detected by the heuristic, the workload is executed for each available core frequency. Each execution is repeated 5 times to reduce statistical error. The standard error is calculated across those 5 iterations. To reduce execution time and provide a flexible development environment the workloads are executed with a simulator. The simulator works on the data traces collected during workload recording and is able to run all 5 iterations in approximately 2 hours on an Intel Xeon E5-1620 processor. Compared to a real time execution on a mobile platform of 222 hours i.e. about 9 days, this is roughly a 94× speedup. Further information about the simulator are presented in the next section.

6.4.1 Interactive Workload Simulator

To allow a fast turn-around and easy access to runtime statistics during LDH development, a mobile workload simulator is used. Existing simulators capable of running a mobile workload like *Qemu* were not considered. Their feature set and simulation accuracy is unnecessarily extensive for the initial steps of the heuristic development. Instead, a new simulator was developed with the only objective of producing an interaction lag profile for a simulated workload. This allows simulation speedups around 100× compared to real time execution.

Based on workload statistics the simulator determines lag starts and uses the LDH to detect the endings. Workload statistics are taken from runtime traces collected during workload executions for the experiments in Chapters 4 and 5. Those data traces contain timestamped information on screen refreshes, input events, if the CPU is busy or idle, current CPU frequency and so forth. With this data, LDH functionality can be tested for a workload. Figure 6.9 demonstrates the simulator's workflow in more detail.

The simulator core runs the main simulation loop and regularly updates a system timer. The timer considers workload time on a millisecond accuracy. For each run of the update loop, the timer provides a new time stamp which is passed on to a System Environment Manger ①. This component has access to data traces of the simulated workload. When it receives the current time stamp, it extracts necessary data at that

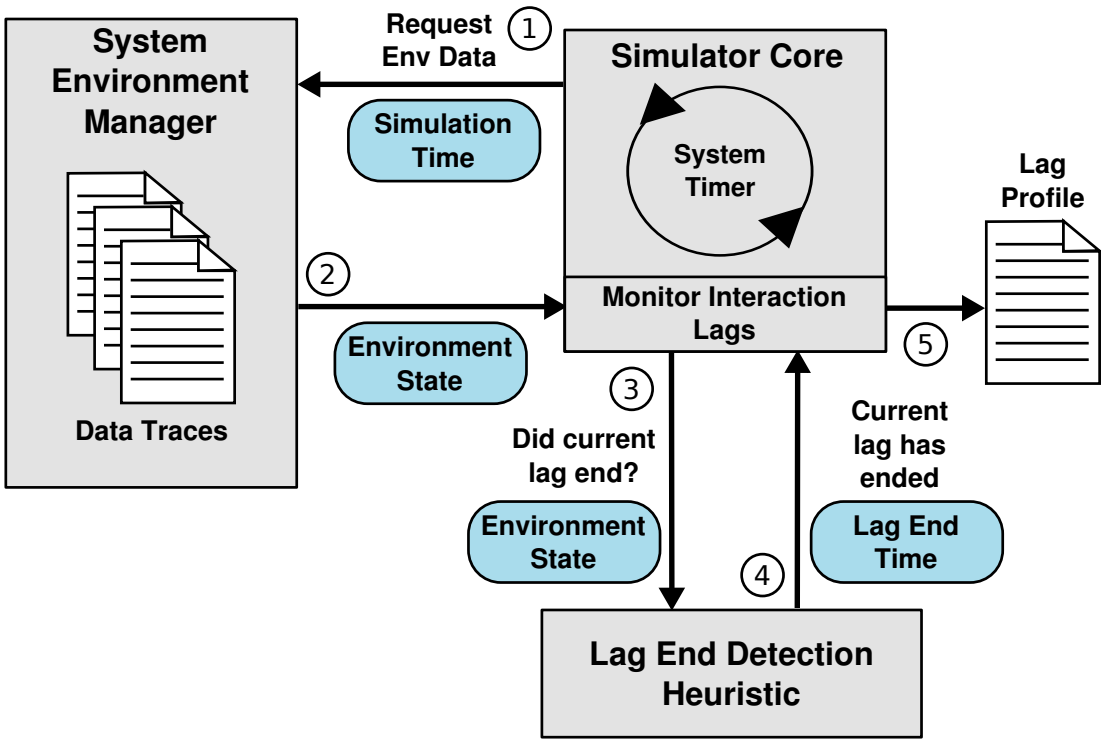


Figure 6.9: Workload simulator workflow of a single update loop. ① A system timer sends the current simulation time to the System Environment Manager. ② This component generates the current state of the system environment based on system traces collected during workload execution on hardware. ③ Based on observed input the core decides if a lag began and starts polling the LDH for the corresponding lag ending. ④ When LDH detects the lag ending, the corresponding time stamp is reported back to the core ⑤ which adds the final lag dimensions to an ultimately produced lag profile.

time from loaded traces and returns the current state of the system environment to the simulator core ②. This state contains information on which statistics collected data traces are showing at that time.

The Simulator Core processes the information and decides if a new lag has started, based on observed user input. While a lag is active, the core regularly polls the LDH for whether the active lag ended yet. To make this decision, it passes the environmental state on to the heuristic ③. When a lag end was detected, the corresponding lag end time stamp is reported back to the Simulator Core ④. The core then ends the currently active lag and saves simulation results in a lag profile ⑤. Like the lag profile generated by the video frame markup method from Chapter 4, the lag profile produced by the simulator contains information on when each lag started and when it ended. Lag profiles of both methods can then be compared to evaluate LDH accuracy. The results of experiments where this was done, are presented in the following sections.

A limitation of the simulator is its inability to simulate nondeterminism caused by background processes of the OS or other applications. Since processing resources of foreground and background tasks are shared during execution, differences in execution statistics can appear for multiple runs of the same workload. To compensate for this limitation, the workload generated in Chapter 4 is executed on actual hardware 5 times and 5 sets of execution statistics are collected. Experiments conducted using the simulator are always repeated for each of those 5 iterations and results are averaged.

6.4.2 Evaluation Metric

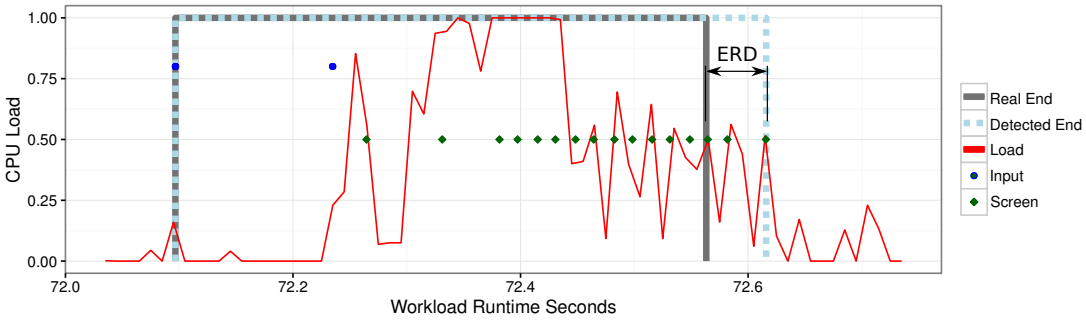


Figure 6.10: This figure shows the error value used to evaluate LDH accuracy. ERD is the percentage error between the real lag ending as measured by the lag marker method in Chapter 4 and the lag ending detected by the LDH developed in this chapter.

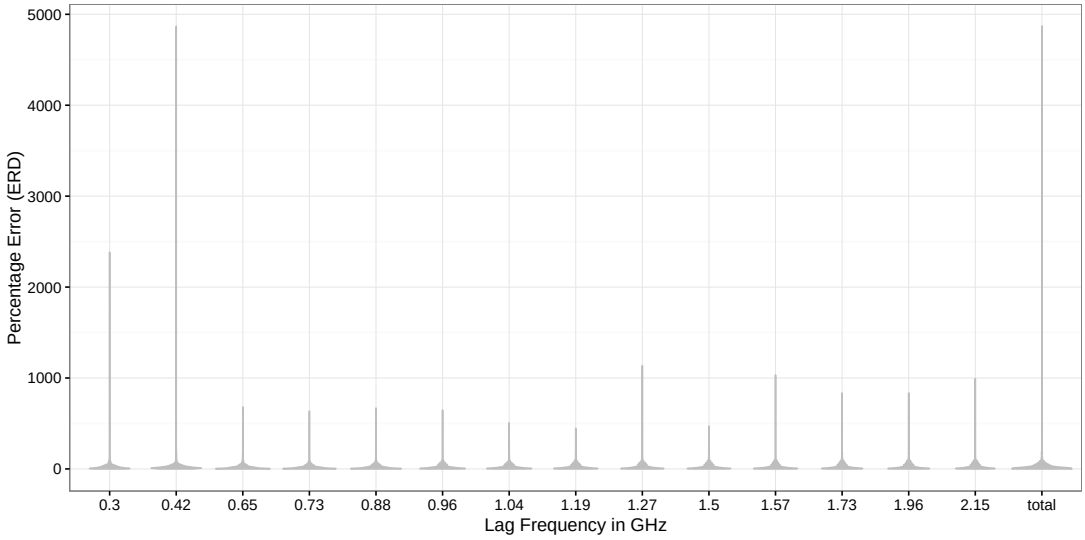
The LDH is evaluated in terms of detection accuracy. For that purpose the error is considered between the lag ending as detected by the video frame markup method, i.e. as seen from a user's point of view, and the ending as detected by LDH. It is called error between real and detected ending (ERD) and displayed for an example lag in Figure 6.10.

The mean absolute percentage error (MAPE) is used as a metric to calculate the actual error value. It is computed by taking the geometric mean of the percentage errors for a set of n lags with lag id i based on the difference between the lag duration with the real lag ending r and the lag duration with the detected ending d. MAPE is defined as:

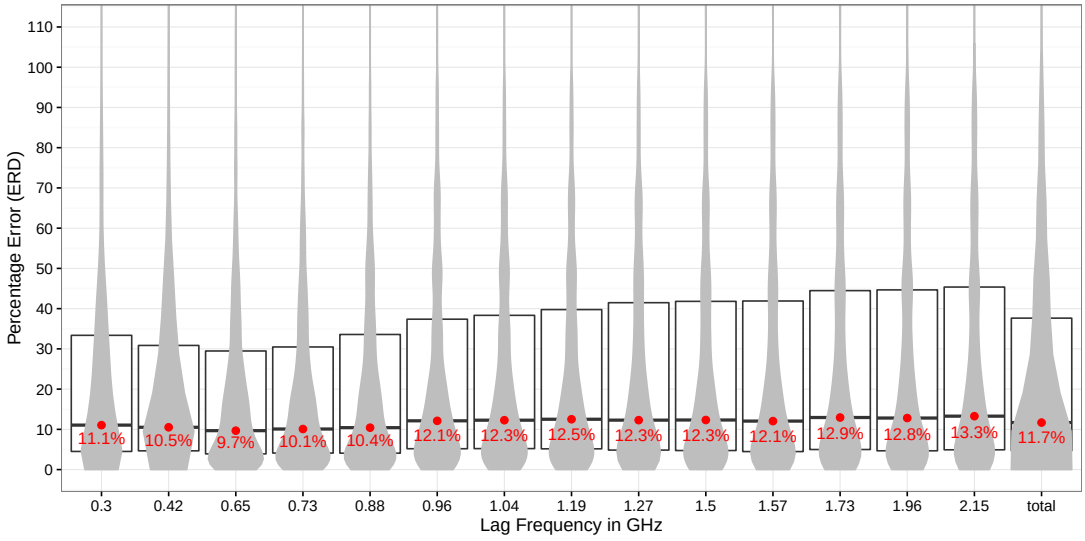
$$MAPE = e^{\left(\frac{1}{n}\sum_{i=1}^{n}\ln\left(ERD_i\right)\right)} = e^{\left(\frac{1}{n}\sum_{i=1}^{n}\ln\left|\frac{r_i - d_i}{r_i}\right|\right)} \tag{6.1}$$

6.5 Experimental Results

As described in Section 6.2 the LDH must be able to find a lag ending for lags of variable length and CPU frequency. Therefore, the complete workload is executed once for each available CPU frequency. During execution the current frequency is fixed for each lag and a medium frequency of 1.04 GHz is used for idle periods between lags. Using a different idle frequency does not affect the heuristic's accuracy since it is only set after the lag ending was already detected.



(a) Distribution of all data points including outliers.



(b) Zoom in on the 1.5×IQR of the displayed data points. Box plots in the background show MAPE (red dot and percentage value) and upper and lower quartiles.

Figure 6.11: Distribution of percentage error between real lag ending and detected ending over all available CPU frequencies. A total distribution is shown on the right.

Figure 6.11 shows the distribution of percentage errors between durations with real and detected lag endings over all available CPU frequencies. The distribution of values is indicated by violin plots for each frequency. Figure 6.11a shows data points for the entire workload executed 5 times for each available CPU frequency. The figure shows some large outliers, especially for the lower two frequencies. A zoom in on 1.5× the interquartile range (IQR) of the data is shown in Figure 6.11b. 5.1% of the data points lie beyond 1.5×IQR. The lag duration MAPE for each frequency is displayed as a red dot and a percentage value. A box plot behind each violin plot shows MAPE and upper and lower quartiles. On the very right, the displayed distribution of percentage errors considers data points from all frequencies. The MAPE values shown in the distribution figure range from 9.7% at 0.65 GHz up to 13.3% at 2.15 GHz with a total value of 11.7%.

6.5.1 Belated Lag Detection

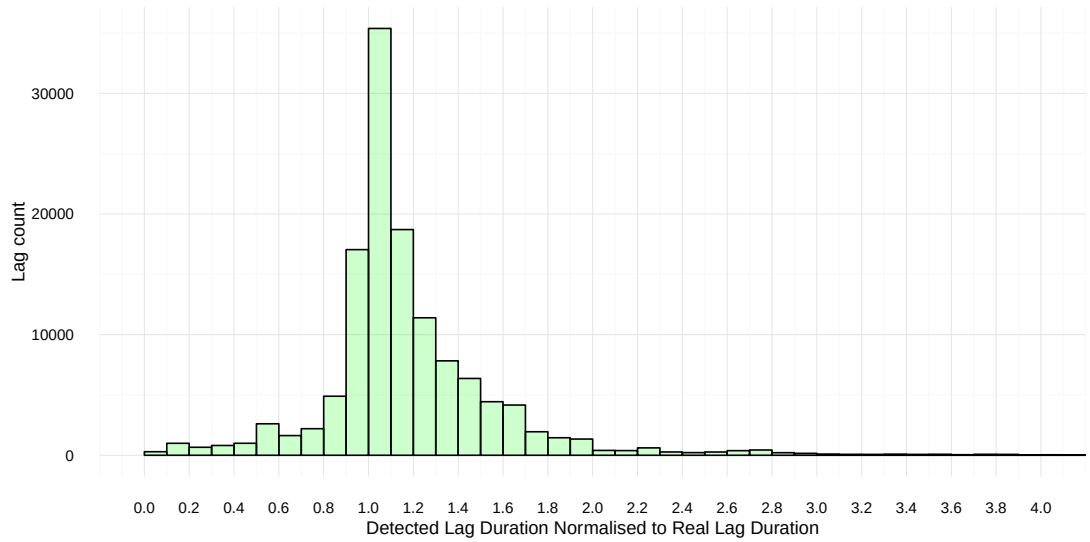


Figure 6.12: Distribution of lag durations with detected lag ending normalised to lag durations with real lag ending.

The distribution of detection error shows that the heuristic is often able to detect the lag ending close to what the video frame annotation method reported. However, it also fails to do so for some cases. Most of those cases where the heuristic gets it wrong, the detected ending lies after the real ending already happened. This is depicted in Figure 6.12. This figure shows a histogram with the distribution of lag durations with the detected ending normalised to the corresponding lag duration with the real lag

ending. It can be seen that more detection errors are larger than one which means the heuristic detected the ending later than the video frame method. The usual cause of belated end detections are screen refreshes stretching beyond the real lag ending. For smaller errors this is due to inaccuracies of how the video frame marker method compares frames which causes it to miss small colour differences (see Section 6.2.1). Larger errors happen when discrepancies exist between screen refreshes and actually visible screen changes (see Section 6.2.4). In those cases screen refreshes continue without causing frame changes visible to the end user.

There are also a few substantially large errors, especially for the two lowest frequencies. They are caused by two types of lags which prevent the heuristic from detecting a lag ending at all. In those cases the next interaction starts before the heuristic reports an ending. As a fail safe, the start of the next interaction is automatically set as maximum end value for a lag. Depending on the length of the idle period following a lag, this can lead to very large errors. The first type of lag where this happens are those which never reach the expected interaction lag ending due to a low frequency. That means, the user never sees the result he expects as a system response when interacting with the device. For example, he taps on a home screen shortcut and starts up an application. The low frequency causes a long load time and ends with the application being loaded missing important interface elements. This can be interpreted as a bug in the application code which is unable to handle extremely low frequencies. The second type are lags which show a visible end to the user but continue with highly CPU intensive background processing. Hence the heuristic, which is waiting for the load to settle after a screen refresh burst (see Section 6.3), never finds such a spot and keeps looking until the next interaction event is issued.

6.5.2 Early Lag Detection

Figure 6.13 shows for which percentage of total lags the ending is detected after the real lag end happened. This value totals to 75.2% over all frequencies. It is as high as 88.2% for the lowest and goes as low as 67.2% for 1.57 GHz. In general, the heuristic detects the lag end before the real ending more often for higher CPU frequencies. This is due to the heuristic waiting for CPU load to settle after a screen refresh burst before reporting a lag ending. For higher frequencies an ending can be detected even though the real lag is not over yet. This happens when a lag consists of multiple screen refresh

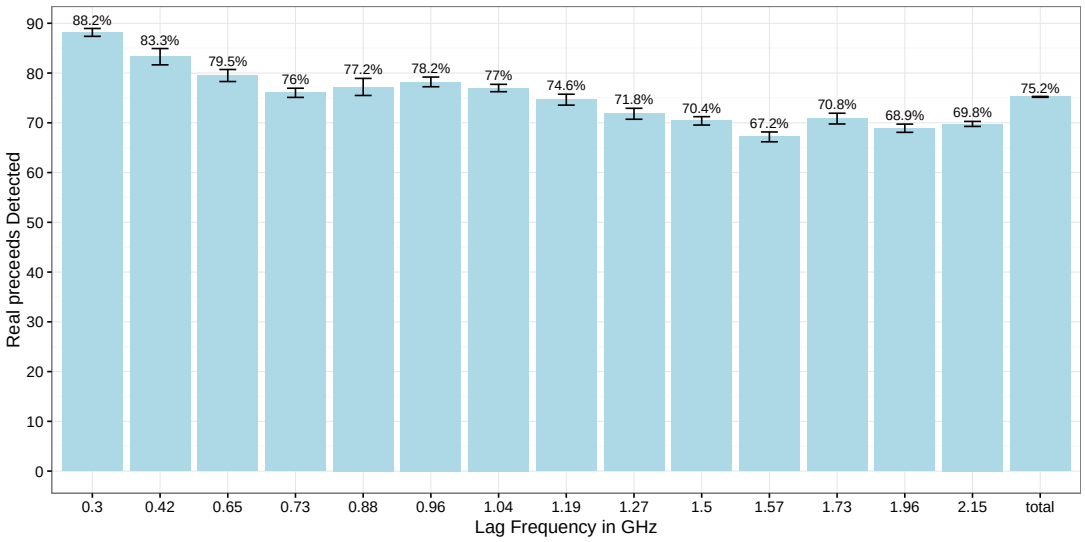


Figure 6.13: Relation of real lag ending to detected one over the complete workload executed with the CPU being fixed to each frequency. The bars indicate when the heuristic detects the lag ending after the real ending already happened for a given frequency setting. Error bars show the standard error which is calculated over 5 iterations of the workload execution.

Figure 6.14: Sample workload of an early lag end detection. Low load and a pause after a screen burst lead to a false positive ending report by the heuristic in the middle of the lag.

bursts and all CPU work related to one burst is done before the next burst started. That means, CPU load drops in the middle of the lag which the heuristic can interpret as an ending.

Figure 6.14 demonstrates this type of premature ending. The lag is a tap on the At-

tractions menu entry in the Stay application. It opens a list view with multiple city attractions. Underlying system statistics are separated into two CPU load bursts. The first one happens while the menu entry is highlighted and the list view fades in. The second one happens while all list entry header pictures are being loaded and fade in one by one (the very first entry does not have a picture). Between the two bursts CPU load goes down to nearly zero. Also screen refresh events pause for a few milliseconds. Now both conditions for the heuristic to report a lag ending are met so the ending is considered detected. On a lower frequency CPU load would be higher than the threshold marking a lag ending, and the gap between the two bursts would be bridged. Problems of this kind can be fixed by tweaking the heuristic for higher frequencies. For example by stretching the timespan for which the heuristic needs to observe low load after a screen refresh burst before reporting a lag end.

6.6 Conclusion

In this chapter a heuristic was developed to capture user perceived workload periods at runtime. By monitoring system events with a strong correlation to visual screen output, this heuristic is able to detect the lag ending of an interaction as seen by the user. The LDH is able to detect lag endings for all CPU frequencies with an average error of 11.7%. It works best in the medium range of the available frequency spectrum. Large detection errors can happen when the *SurfaceFlinger* screen refreshes used by the heuristic do not lead to frame changes visible to the user. In those cases the lag ending is detected too late. Another frequent cause of errors are periods in interaction lags where the CPU load is nearly zero for some time due to animation timers, memory stalls or disk I/O. Here the heuristic can report a lag ending too early. In 75.2% of the cases the detected ending lays after the ending as seen by the user. In the next chapter the heuristic will be used to develop a CPU frequency governor. It will be shown that the heuristic's detection accuracy is sufficient for the developed governor to achieve improved energy efficiency and user irritation and that the governor is able to compensate for potential detection errors.

Chapter 7

QOE Driven DVFS Algorithm

7.1 Introduction

Research experiments conducted in this thesis are driven by the claim that a DVFS technique for interactive mobile workloads can accomplish good energy efficiency if information is available on how fast the system responds to interactions as seen from the user's point of view. A methodology and metric to benchmark a representative mobile workload in terms of QOE were developed in Chapter 4 and Chapter 5. Existing potential for energy savings whilst maintaining high QOE was shown in Chapter 5 by comparing frequency selections of current standard frequency governors with an all knowing *Oracle*. Chapters 6 and 7 are dedicated to developing a QOE aware DVFS technique which is able to closely match the *Oracle*.

In the previous chapter a lag end detection heuristic (LDH) was developed which allows the desired DVFS technique to subdivide an interactive workload into lag and idle periods at runtime. The LDH will be used in this chapter to give the QOE governor the necessary information on the user's perspective of the workload. With that information, the governor needs to find the frequency with optimal energy efficiency and QOE for each lag and idle period encountered during execution. Optimising energy and irritation for idle periods is solved in a straightforward way. Irritation is per definition zero and a hard-wired CPU frequency provides good energy efficiency for most cases. Finding a good frequency for interaction lags, however, is more challenging. It is achieved with machine learning.

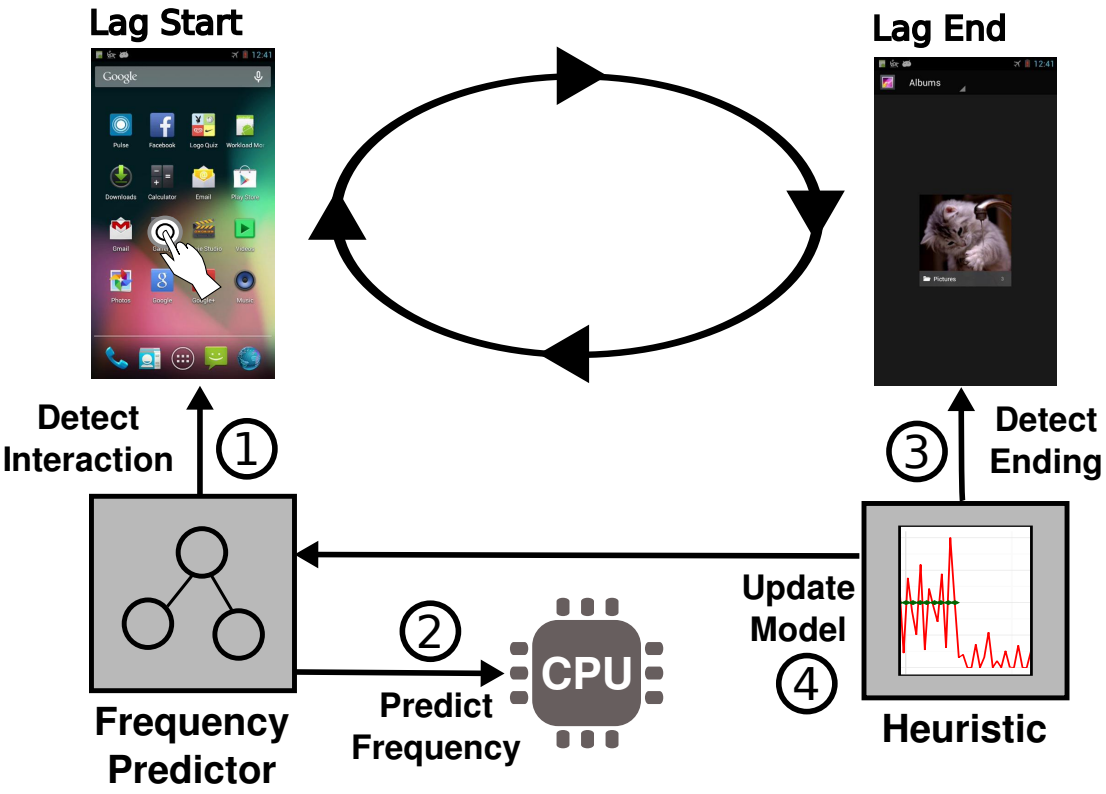


Figure 7.1: High level concept of QOE aware DVFS technique. At ① the governor detects an interaction. It predicts an optimal frequency at ②. The interaction executes until its end is detected at ③ by the heuristic from Chapter 6. The lag's measured duration is reported back and used to update the frequency prediction model at ④. Observation of multiple samples of an interaction improve frequency prediction over time.

7.1.1 Machine Learning Driven DVFS

Finding a good frequency for an upcoming interaction lag at runtime is extremely hard without prior knowledge of the lag's behaviour. For specific applications it might be possible to anticipate behaviour, but not for the general case. An often used technique for such problems is machine learning. By observing multiple interaction samples, lag behaviour for different frequencies can be learned. The acquired knowledge can be used to train a model to predict runtime decisions. A static model, however, would need constant adaptation with new training data for each new application released and each new device published.

The approach taken in this chapter to find good frequencies for lags is based on reinforcement learning (RL). RL applies a trial-and-error approach to find an optimal solution in a dynamic environment. A feedback loop provides updates on executed

behaviour and reinforces good results to constantly adapt and improve the underlying model (see Section 2.4 for details on RL). The approach is based on the assumption that the same interactions or the same type of interactions appear over and over during workload execution: For example, typing keys on an on-screen keyboard, regularly opening the email application to manage messages, or playing a game with a given set of controls. The idea is to learn good frequency levels for recurring interactions by observing execution statistics of multiple samples while trying different frequencies.

Figure 7.1 shows how the QOE aware governor applies the RL technique. As soon as user input arrives, the interaction is identified ① and a QOE model predicts an optimal frequency for the upcoming lag. The CPU is set to this frequency ②. When the user perceived ending is reached, LDH detects it and reports lag duration ③. The measured duration and calculated energy consumption are used to update the QOE model and to improve frequency prediction for the next sample. By observing the results of frequency selection for multiple interaction samples, the governor constantly adapts prediction accuracy. This way it can find good energy efficiency while maintaining low user irritation. The name Reinforcement Learning Governor (*RLGov*) will be used in the remainder of this thesis.

Other DVFS techniques were developed using RL [174, 175]. They did, however, focus on batch workloads and not interactive ones. *Li et al.* [177] introduce a supervised learning based DVFS technique for mobile workloads. They train a neural network to predict a frequency setting on application granularity and rely on user questionnaires to improve their QOE model. Considering the size of modern smartphone applications it is unlikely that performance settings on application granularity are sufficient to achieve high energy efficiency. Furthermore, improving the prediction by relying on user feedback is likely not practical in a realistic scenario (see Chapter 3 for details).

7.1.2 Identifying Interaction Events

To learn good behaviour for specific interactions, *RLGov* needs to identify the type of the upcoming interaction lag ahead of time. This way multiple small QOE models for each encountered interaction type can be maintained instead of a large complex one for the general case. This approach reduces feature space and increases frequency prediction efficiency. Since the research focus of this chapter is on predicting optimised frequencies, interaction identification is done manually for conducted experiments.

Nonetheless, techniques for automatic identification are discussed in Section 7.3.

For 66% of the interaction types used in this study *RLGov* settles on a frequency prediction with a good tradeoff between energy and irritation after a training period of 14 execution samples. This statistic goes up to nearly 90% after 60 samples. After the initial learning phase the governor achieves an energy consumption of 9.6% above the *Oracle* profile calculated in Chapter 5. This consumption is up to 22% lower than the energy consumption of current standard *Android* frequency governors. The user is irritated for 5.6% of the total interaction lag time of the benchmark workload.

7.1.3 Contributions

The contributions of this chapter are:

1. A runtime reinforcement learning based technique to predict the CPU frequency with optimised energy efficiency and QOE for an upcoming interaction lag,
2. a runtime technique to identify user interactions ahead of time,
3. a runtime model to quantify user irritation caused by interaction lag, and
4. a reinforcement learning based frequency governor for interactive mobile workloads with improved energy efficiency and QOE compared to current mobile solutions.

7.1.4 Overview

Section 7.2 will present the overall concept of the QOE aware governor. Section 7.3 will describe how interaction events can be identified ahead of time. This is followed in Section 7.4 by a description of how *RLGov* uses an RL approach to learn execution behaviour of interaction lags. The reward function *RLGov* uses to optimise frequency choices will be explained in Section 7.5 and the frequency selection process for each upcoming lag in Section 7.6. In Section 7.7 the experimental setup will be presented to evaluate *RLGov*'s functionality. Experimental results are described and discussed in Section 7.8. Lastly, Section 7.9 will summarise and conclude the chapter.

7.2 Reinforcement Learning Governor Concept

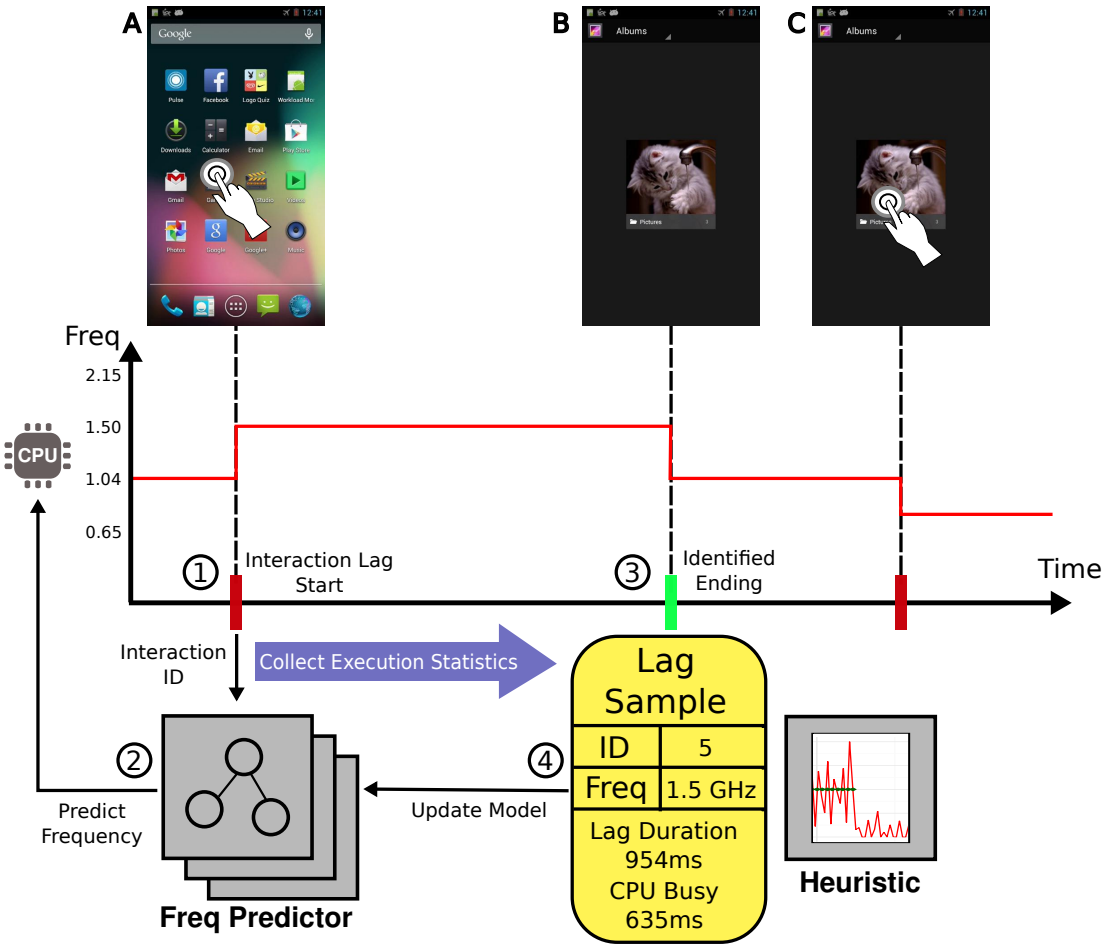


Figure 7.2: Concept of *RLGov* handling an example interaction of opening the Gallery application: ① Interaction identification as soon as user input is detected, ② frequency prediction for the upcoming lag from the corresponding model in its current training state, ③ lag end detection as seen by the user and ④ switch to idle frequency and updating the prediction model with collected execution sample statistics.

Figure 7.2 shows a more detailed concept of *RLGov*'s frequency prediction for an interaction sample. A timeline of the execution goes from left to right. The screenshots A to C above the timeline show the screen output of three distinct execution events: Interaction lag start A, user perceived ending of the lag B and start of the next interaction lag C. Below the screenshots is a graph indicating which frequency is set to the CPU. Below the timeline a flow chart shows how *RLGov* makes frequency predictions. The executed interaction is a tap on the homescreen shortcut of the Gallery which leads to the system response of starting this application. The following paragraphs will explain frequency selection for the two user perceived workload periods.

Lag Frequency When the system receives input at A the interaction is identified (see Section 7.3) and a corresponding id is generated. The id is passed on to the frequency predictor ①. To improve prediction efficiency, the frequency predictor maintains a model for each encountered interaction id. It picks the associated one and makes a frequency prediction based on previous observations ②. The prediction aims for the frequency which results in the lowest tradeoff between energy consumption and user irritation for the encountered lag among all available frequencies. The tradeoff is calculated by multiplying measured energy and irritation. The prediction accuracy is improved by observing execution results following an interaction and using them to update the prediction model. The CPU is set to the predicted frequency and lag execution starts. During execution *RLGov* monitors system statistics and uses the heuristic developed in Chapter 6 to find the ending of the interaction lag as seen by the user. When the heuristic reports the lag end, the observed statistics are recorded ③. The observed execution statistics are considered a sample of the encountered interaction type using the predicted frequency. Based on the sampled statistics, energy consumption and user irritation of the executed lag are calculated as well as their resulting tradeoff. These values are used to update the prediction model associated with the id generated for the interaction ④. Thereby, the frequency governor is able to optimise its behaviour for previously unseen interactions and adapt to changes in behaviour of known ones.

Idle Frequency As was discussed in in Section 4.2, the CPU frequency of the user perceived idle period between B and C does not affect QOE. Here, the frequency governor can optimise for energy alone. Since during most of the idle period the CPU is not busy, a CPU frequency with the best overall energy consumption for idle periods is hard-wired. This approach is a good solution for most idle times. Always choosing the most energy efficient idle period would require additional effort. Knowledge would be needed of how long the idle period is going to be and which potential background work would occur. Among other features, this knowledge depends on a human component, e.g. the user's decision to continue interacting with the device. It would require a different approach and is therefore left for future work. It will be shown in the experimental section that using a hard-wired frequency leads to an energy consumption of 13.4% above the *Oracle*'s output. This is up to 29.2% lower than what current standard frequency governors use on *Android*.

The two main features energy and irritation used to drive prediction need to be determined efficiently and accurately at runtime. Energy is calculated by using a CPU power model as was done for the video markup method. The offline irritation metric from Section 5.2 was adapted for online application in *RLGov* since a comprehensive knowledge of execution data can no longer be assumed at runtime. It generates a value for user irritation based on various runtime statistics. The lag end is detected by LDH which considers the correlation of execution statistics to the visual output of the system. Among those statistics are CPU load and screen refresh calls by the *Android* framework module *SurfaceFlinger*. The following sections will firstly describe how interactions can be identified ahead of time to maintain a prediction model for each. Secondly, the functionality of *RLGov*'s components will be explained in detail which is eventually followed by an evaluation of its feasibility.

7.3 Identifying Interaction Events

This section will introduce the concept of different interaction types and discuss techniques to identify user interactions at the start of a lag. This way *RLGov* is able to maintain small and efficient prediction models for individual interactions instead of a complex one for the general case.

7.3.1 Identifying Interactions

The goal of identifying interaction events is to be able to distinguish ahead of time between different interaction lags with different execution behaviour. The entry point of the executed system response following a user interaction is the input handler function of the current foreground application. This function is responsible for handling the received user input. In this thesis, only touchscreen input is considered. Different handler functions are tied to different UI elements and actions such as buttons, text input fields, focus changes or game control input. Each application has its own layout tree specifying the positions and dimensions for each UI component. The ideas behind identification methods proposed in other research are based on instrumenting the mentioned application or OS components responsible for handling an interaction. Application or OS instrumentation can then supply an interaction identifier calculated from the program counter of the corresponding handler method, the name of the handler's

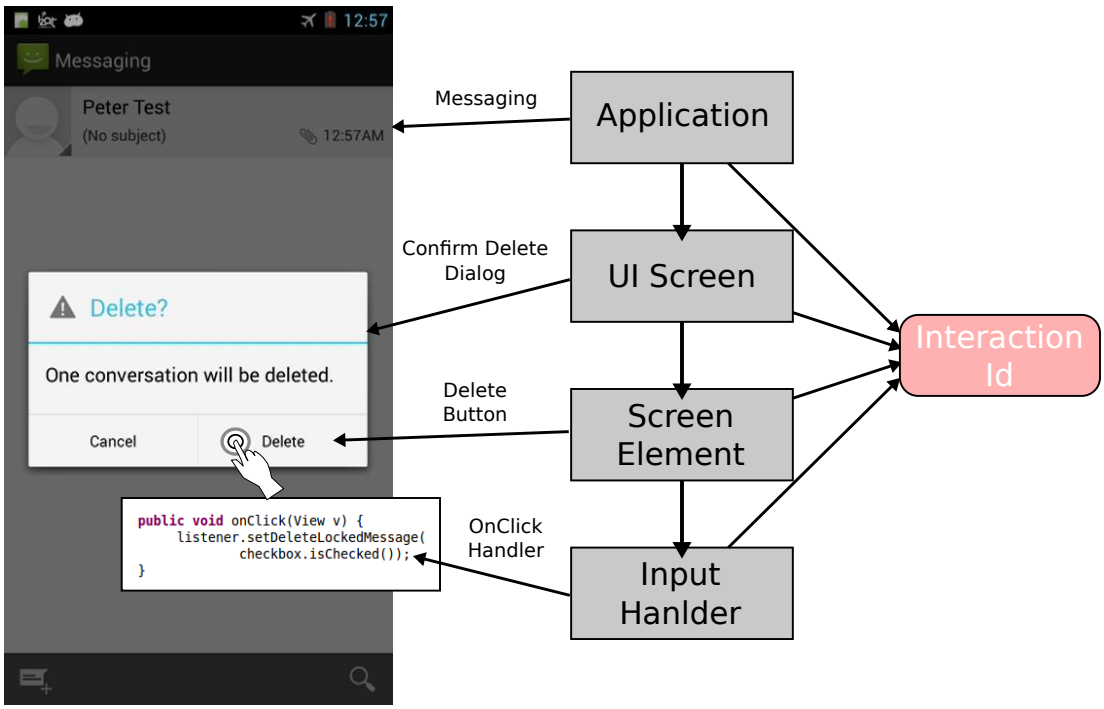


Figure 7.3: Input event handler chain for an interaction example. Identifiers of components participating in handling the given input can potentially be used to generate an interaction id.

class or the name of the element in the layout tree. The actual elements responsible for handling an interaction depend on touch coordinates and type[1] of the input and on the currently visible screen of the foreground application.

Figure 7.3 demonstrates this identification concept for an input example. The left side shows a screenshot of the current state of an *Android* mobile device. A finger indicates where the input event happens. The input confirms a dialogue which asks to delete a conversation in the messaging application. The centre of the figure shows the corresponding chain of framework and UI elements involved in handling the input. At its start is the application itself. It is followed by the UI screen showing the *Confirm Delete* dialogue. The screen element receiving the input is the delete button which is tied to an underlying *onClick* handler method. Identifiers of all participating elements such as class names, method names or string identifiers can potentially be used to identify the interaction ahead of lag execution time.

Actual implementations can be found in a publication of *Song et al.* [168]. They capture user perceived SRT periods for *Android* workloads by using *Dalvik* virtual machine instrumentation. Each user input in *Android* is processed by the underlying

[1]Types are for example tap, long touch, swipe, pinch, etc.

framework and passed on to the foreground application. An input interface used in the framework provides different handler methods to handle different input types such as *onClick()*, *onSwipe()*, *onLongClick()* and so forth. These methods are instrumented to track interactions. Another implementation is presented by *Zhu et al.* [152]. They track interactions in the *Android* web browser. For that purpose the web browser sources are instrumented to monitor its internal interaction event queue. Each event in that queue is tied to a corresponding handler which is used to identify interactions. *QoEDoctor* [9] is a tool to monitor SRT durations in interactive workloads. It uses an instrumentation of application layout trees to identify interactions. Further information on these publications can be found in the related work in Chapter 3.

7.3.2 Interaction Class and Instance

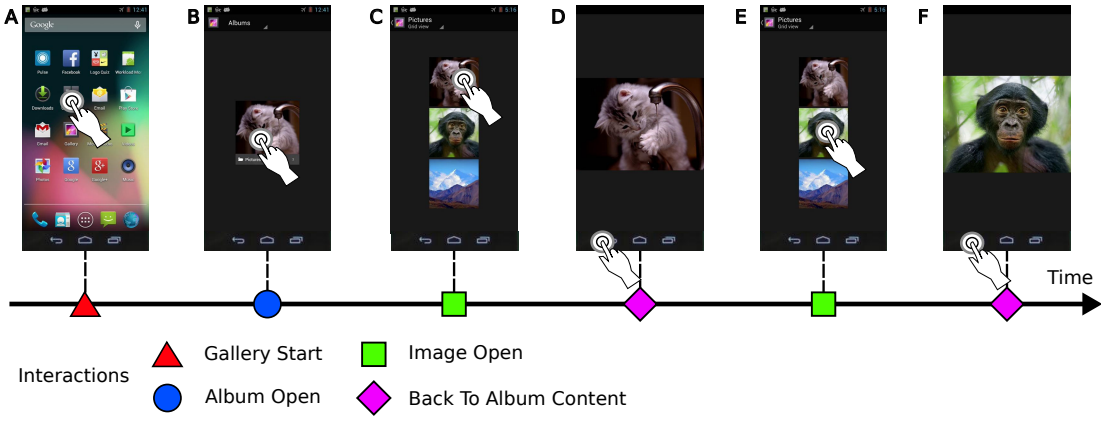


Figure 7.4: Workload sample showing multiple interactions with corresponding interaction identifiers, which are represented as geometrical forms. When interactions are similar it can happen that they are associated with the same id, like *Image Open*.

Figure 7.4 is showing an example of successive interactions where different inputs are provided which lead to different interaction events. On the bottom is a timeline showing generated interaction ids represented by different geometrical forms and colours. All occurring interactions are listed below the timeline. Screenshots above the timeline show the screen output belonging to each executed input and a finger marks the spot where the input happened. Six input events occur in the displayed workload snippet. The first event is a tap on the Gallery shortcut on the home screen, which opens the Gallery application. The next event opens a picture album in the Gallery. Then, a picture is selected and the back button is pressed to return to the album content. Finally, a different picture is selected and the back button is pressed once more. Even though

there are six input events only four interaction identifiers are listed on the bottom: *Gallery Start*, *Album Open*, *Image Open* and *Back To Album Content*.

It is possible that the handler chain is the same for different interactions. For example, clicking on two different images is handled by the same handler method and corresponding UI elements are not distinguishable. Depending on the instrumentation method, the two interactions can receive the same identifier for the given case. It is, however, likely that interactions which are handled by the same handler method exhibit the same execution behaviour. If so, *RLGov*'s learning method would not be negatively affected. In fact, learning the corresponding behaviour would happen faster since more samples are provided to the governor. However, if two interactions with different execution behaviour end up with the same identifier, *RLGov* would have difficulties optimising frequency since inconsistent sample data would be provided. In the worst case the frequency determined best by the governor would be suboptimal for both types of behaviour profiles. An example for bad interaction type classification and how *RLGov* handles this case is presented in Section 7.6.4.

The terminology used in the remainder of this chapter will be as follows:

Interaction Class represents a group of specific interactions which are given the same identifier.

Interaction Instance is an actual occurrence of an interaction from a certain interaction class during workload execution.

The presented technique can be used to allow *RLGov* to identify upcoming interaction lags ahead of time based on the given input and screen state of the running application. In so doing, *RLGov* can keep multiple small prediction models, one for each encountered interaction class. This reduces feature space and complexity and increases prediction efficiency. Presented identification approaches do not require application developers to modify or reveal their code.

7.4 Learning Good Frequency Choices

To optimise the CPU frequency for lags of a single interaction class, *RLGov* uses a reinforcement learning (RL) based approach. An RL learning problem is broken down into multiple situations where the RL algorithm needs to make a decision on which

action to take. The algorithm learns good behaviour by observing the results following its action. Observed results are condensed into a single number called a *reward*. The higher the reward, the better the executed action. For the given problem, decision making situations are the beginnings of lags. Behaviour choices are selecting an available CPU frequency. The observed results are energy efficiency and user irritation measured over the lag period. They are condensed into a single tradeoff by multiplying both parameters. A minimum tradeoff maximises the reward and is therefore *RLGov*'s learning goal.

7.4.1 Slot Machine Analogy

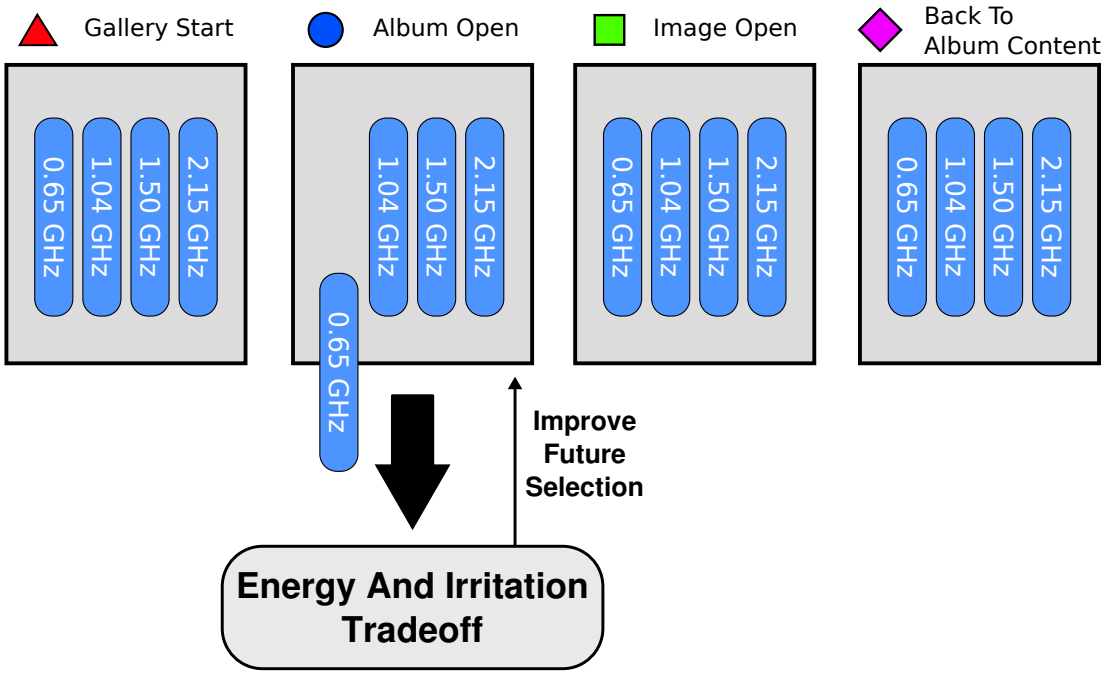


Figure 7.5: Frequency prediction models depicted as slot machines. Frequency selections are indicated by different levers. Selecting a frequency yields a reward of energy and irritation tradeoff after lag execution. This reward is used to improve the frequency selection of the corresponding model.

In the given workload, frequency selections for single lags are independent of each other. Selecting a certain frequency for the first lag, does not influence which frequency needs to be selected for the second one and so forth. This simplified case of an RL problem is called a *Multi-Armed Bandit Problem* (see Section 2.4 for details on RL). Its name comes from the colloquial term "one-armed bandit" used for slot machines in casinos. An often used analogy describes a single slot machine with multiple levers.

Each play allows pulling a single lever and the reward is the amount of money won. Each lever has a different payout rate and the goal is to maximise earnings by learning which lever works best.

For better understanding, an application of this analogy to the frequency selection problem is depicted in Figure 7.5. As an example the interaction classes from the earlier Figure 7.4 are considered. *RLGov* learns good behaviour for each interaction class independently. Therefore, in the figure, each class is represented by its own slot machine. Each machine has four levers indicating four frequency choices[2]. Pulling a certain lever, i.e. selecting a frequency for an upcoming lag, yields the two execution statistics energy consumption and user irritation. In the example, an interaction instance of opening an album in the Gallery is encountered and the frequency 0.65 GHz is selected. Resulting energy and irritation are combined to a tradeoff as a reward value for that slot machine. It is used to optimise the selection process for the next encounter of an instance of the same interaction class. How the reward is calculated from collected statistics is explained in the next section.

7.5 Reward Function

Each RL algorithm uses a reward function to calculate a reward value from observed results. *RLGov*'s reward function is displayed in Figure 7.6 for a single interaction class behaviour model. After a specific frequency was selected for an upcoming lag, execution statistics are collected in the background until the lag end is reached. The execution statistics, aka. the lag sample, is passed on to the reward function to calculate a corresponding tradeoff. The most important parameters of a lag sample are lag duration as reported by the LDH (see Chapter 6) and the total time the CPU was busy during execution.

A runtime irritation model is used to calculate user irritation. It uses the reported lag duration and *SurfaceFlinger* screen refreshes observed during lag execution. Energy is calculated from CPU busy time and the frequency selected for the lag. A power model of the target platform's CPU is used to accomplish this. More details on the energy and irritation model are presented in the following sections. Energy and irriation are even-

[2]For simplicity only four interaction classes and four frequencies are displayed. In this study's experiments several hundred interaction classes are considered and the used CPU supports 14 frequency settings.

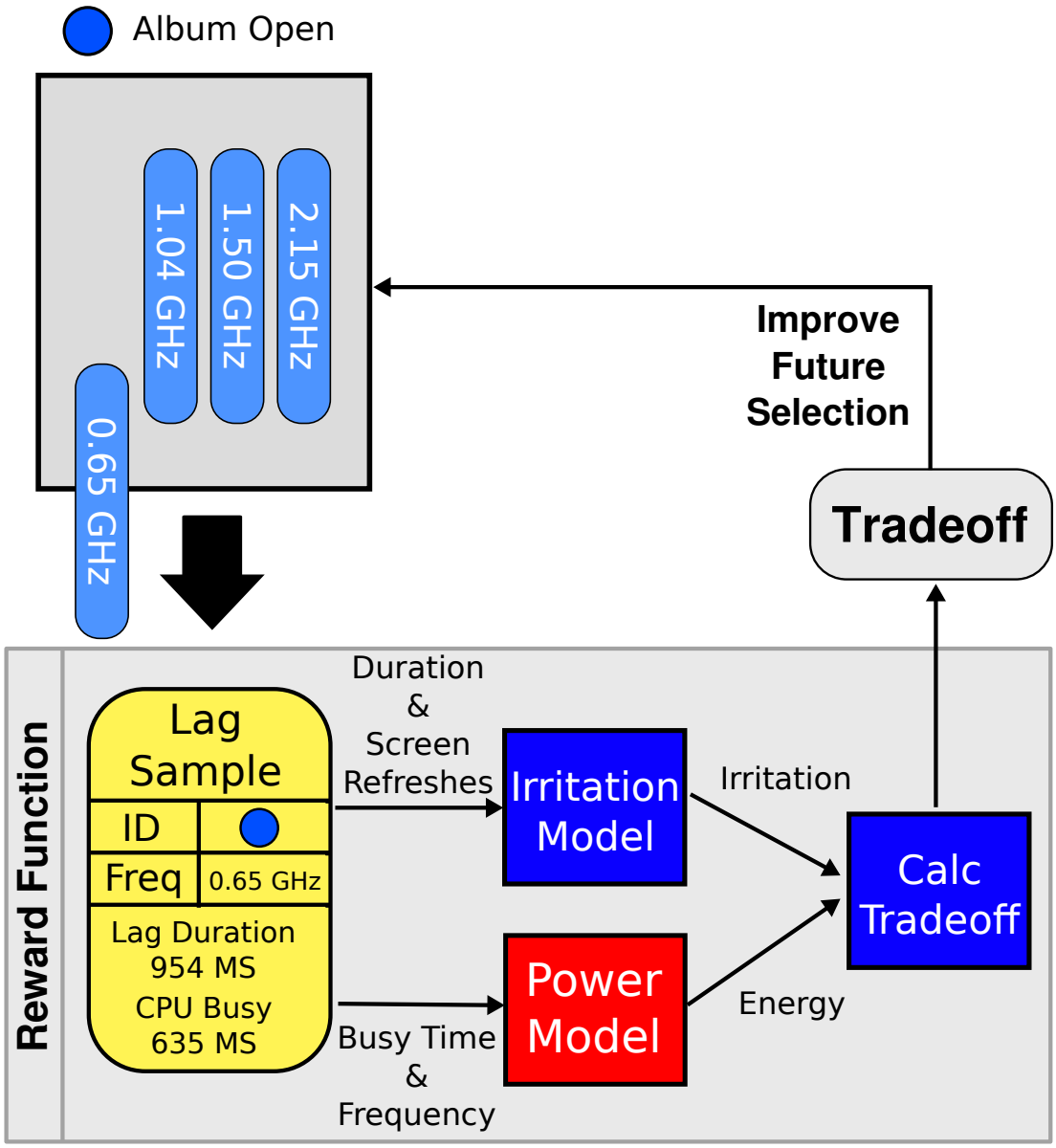


Figure 7.6: *RLGov*'s reward function considers observation statistics sampled during lag execution to calculate a corresponding tradeoff. Lag duration and *SurfaceFlinger* screen refreshes are used to calculate user irritation and CPU busy time and selected frequency to calculate energy consumption. Energy and irritation are multiplied to calculate a tradeoff. The smaller the tradeoff, the better.

tually combined to obtain the corresponding tradeoff. How this tradeoff is calculated can be configured in *RLGov*. Different mobile system vendors may have different requirements on how energy savings and user irritation should be balanced. For the experiments conducted in this chapter, the following expression is used to calculated tradeoff T from energy E and irritation I:

$$T = E \times I^2 \tag{7.1}$$

In the tradeoff equation, irritation is squared for the following reason: While sampling different frequencies for an interaction class and calculating corresponding tradeoffs it can happen that tradeoffs of two different frequencies are close to each other even though the frequency values are not. For example, a high energy consumption but low user irritation can lead to a very similar tradeoff as a low energy consumption and a high user irritation. In this case the design decision was taken that *RLGov* should favour the frequency with the lower irritation value. Higher irritation is instantly noticed by mobile phone users while higher energy consumption reflects in shorter battery life which drains slowly over time.

For each frequency choice of an interaction behaviour model a tradeoff history is maintained. The most recent value including the sampled statistics are added to that history. History contents are considered when making frequency decisions for future encounters of interaction lags belonging to the same interaction class.

7.5.1 Power Model

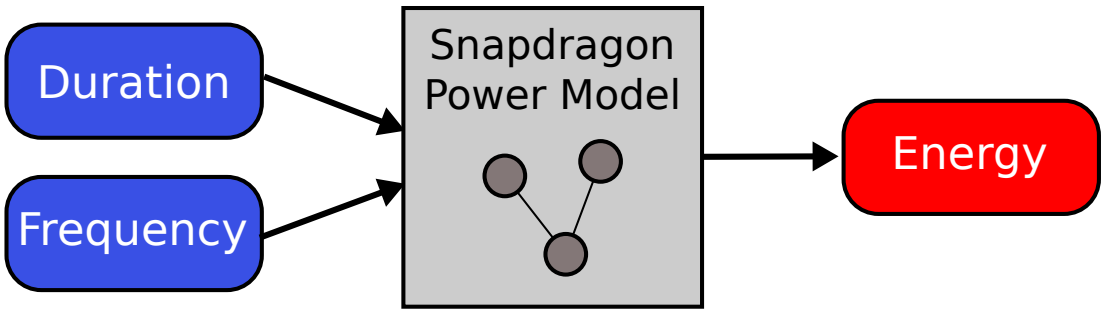


Figure 7.7: Snapdragon Power Model: Power level determined by frequency over time gives energy consumption.

The power model used by *RLGov* to calculate energy consumption is the same as used previously for the experiments in Chapter 5. It was generated by executing CPU intensive micro benchmarks for available core frequencies on the target platform. Idle system power was subtracted from the result to get dynamic core power. When a lag is executed for a certain frequency, *RLGov* records for how long the CPU was busy. Busy time multiplied by corresponding power level results in energy consumption (see

Figure 7.7). The governor is designed in such a way that the power model can be easily swapped for a more sophisticated version. If the hardware provides the necessary sensors, energy could also be measured directly.

7.5.2 Irritation Model

Sampled lag duration is given to an irritation model to calculate a value for user irritation. This value is used by *RLGov* to get an indication of how high or low the user's QOE is depending on the duration of the lag for the given frequency configuration. The irritation model maps elapsed time to a corresponding irritation value by using a Gompertz shaped function graph. The Gompertz function is defined as:

$$y(t) = ae^{-be^{-ct}} \tag{7.2}$$

In this formula, a is the asymptote y values are growing towards. b and c are positive numbers where b sets the displacement along the x-axis and c sets the growth rate. e is the exponential function.

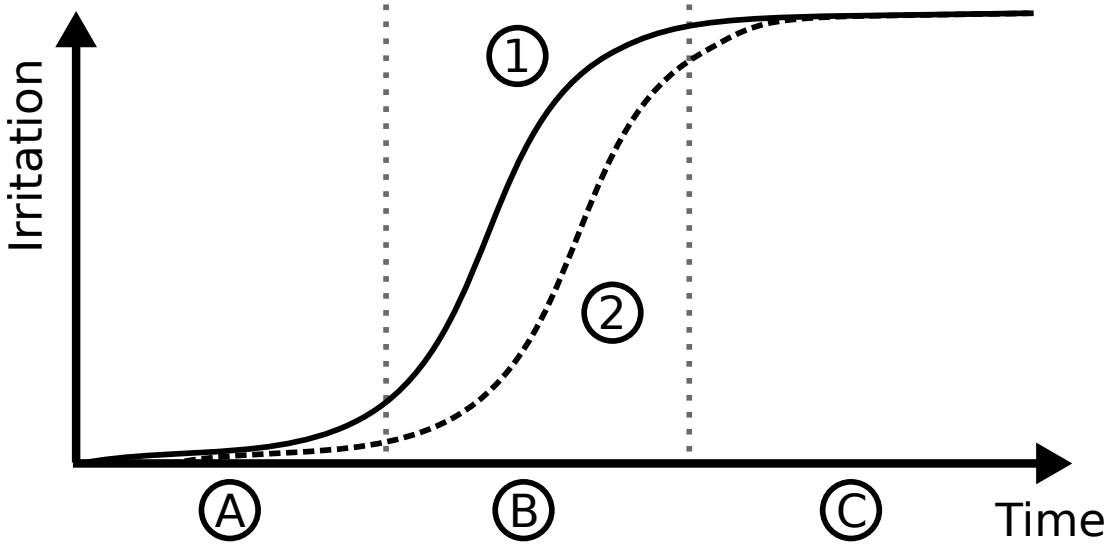


Figure 7.8: Runtime irritation model using the Gompertz function shape. Elapsed lag time is translated to an irritation value. Initial slow grow during Phase Ⓐ is followed by a rapid irritation increase in Phase Ⓑ and runs out towards a maximum in Phase Ⓒ. When *SurfaceFlinger* screen refreshes during execution are considered as reduction of irritation, the graph moves along the positive time axis (① to ②).

The corresponding graph is shown in Figure 7.8 and indicated by a solid line marked with number ①. It is split into three phases: During Phase Ⓐ the irritation value grows

slowly, then changes to a rapid growth in Phase Ⓑ and finally runs out slowly again in an asymptotic manner towards a maximum saturation in Phase Ⓒ. This particular graph shape was taken because it resembles the actual user irritation growth for the kind of interaction lags considered in this study. When the user interacts with the device there is a short time of about 50 to 200 ms where he is not able to perceive any delay at all. This is followed by a similarly long period where he accepts a delay depending on the interaction being executed [11, 12]. Those two time spans make up Phase Ⓐ in the model where the irritation grows slowly. After this initial period the user's irritation grows rapidly the longer the system needs to process an interaction lag as shown in Phase Ⓑ. Finally, irritation saturates at an upper cap in Phase Ⓒ. This saturation is based on the assumption that the user realised that the interaction takes particularly long to be processed and is willing to accept it.

Next to mapping the elapsed time using the Gompertz function, the influence of the screen refresh rate on the irritation growth is considered as additional factor. A widely used technique in the field of interface design and human computer interaction [12] is to give the user something to look at while the system is busy processing his request. This way the user feels that the system is not stuck and is still working to do the job requested. It also serves as a distraction during the waiting time to further reduce irritation growth. Examples can be a loading bar or loading animation, screen transitions, changing pictures or messages, and so forth. To account for this effect, the lag duration given to the irritation model is reduced by a configurable amount for each encountered screen refresh during lag execution. This behaviour results in shifting the Gompertz graph in the positive direction on the time axis. The more screen refreshes are observed in a lag, the larger the shift. Figure 7.8 indicates this shift by displaying a second dashed line marked with number ②.

```
1  function updateIrritationTime() {
2    timeSinceLastScreenRefresh += elapsedTime;
3    if(screenRefreshObservedSinceLastUpdate) {
4      compensatedIrritationTime +=
5        timeSinceLastScreenRefresh * IRRITATION_SCREEN_REFRESH_BONUS
6
7      timeSinceLastScreenRefresh = 0;
8    }
9  }
```

Figure 7.9: Pseudo code algorithm to calculate lag irritation. Lag duration is compensated for screen refreshes encountered at lag execution time. Once the lag is over, the compensated time is passed to a Gompertz graph irritation model to calculate irritation.

The algorithm to calculate user irritation is displayed as pseudo code in Figure 7.9. The function *updateIrritationTime* is regularly called while a lag is being executed. Elapsed time in micro seconds since the last update is accumulated until the next screen refresh is encountered or the lag is over. When a screen refresh is encountered, the accumulated time so far is multiplied by the factor *IRRITATION_SCREEN_REFRESH_BONUS*. For the experiments used in this study, this factor is set to 0.95 which leads to a reduction of accumulated time. At the end of the lag, the compensated irritation time including the time accumulated since the last screen refresh is passed to the Gompertz graph irritation model and a user irritation value is returned.

Measuring and modelling user irritation is a highly complex and extensive research field by itself and large bodies of work exist that look at this topic. The model proposed above is considered to give a good approximation of irritation behaviour for the lags considered in this thesis because it reflects the major findings of HCI studies on the topic of system response time (see Chapter 2.3.1). They are an initial timespan of no to very slow irritation growth because users are unable to perceive a duration difference, the observation that longer lags mostly lead to growing irritation and the acceptance or resignation of a long response time at a certain point. To further increase accuracy, extensions are possible, such as an interaction lag variance factor between different interaction instances of the same class could be added. The higher the variance between two interactions of the same type the higher the irritation. Also a classification of event types as proposed by *Shneiderman* [12] or *Seow* [13] could help to increase accuracy: The user would, for example, expect a shorter delay from light weight events such as key presses on the on-screen keyboard or taking a picture with the camera. Therefore, his irritation would grow a lot faster than it would for more complex events such as application startups or saving an image. Since this is, however, not the main focus of this work, suggested extensions to the model proposed above are left for future work. Like the power model, the irritation model can easily be reconfigured or swapped for an alternative.

7.6 Frequency Selection Policy

A major challenge when solving each RL problem is finding a good balance between *exploration* and *exploitation*. The algorithm needs to decide whether to optimise immediate reward by exploiting gathered knowledge from previous observations or to

explore less ideal choices to optimise future rewards. *RLGov* uses two phases to tackle this challenge for each interaction class. During an initial *Exploration Phase*, the frequency space is explored for the corresponding interaction whilst making sure the immediate reward does not stay low for too long. Once, the underlying model is trained to a certain degree, *RLGov* changes to an *Exploitation Phase*. Now frequencies yielding high rewards are selected. Only occasionally less promising options are explored further to account for potential behaviour changes of the interaction (more on behaviour change in Section 7.6.2). This section will give more detail on the selection process and will eventually show how *RLGov* settles on an optimised frequency after an initial *Exploration Phase*.

7.6.1 Calculating a Selection Weight

Frequency selection is based on sampled execution statistics gathered from previous observations. After each execution of a lag, sampled statistics and corresponding tradeoff are saved in a history. Each frequency of each interaction class has its own history entries. History contents are then used to learn from past behaviour.

Figure 7.10 shows the selection process for a single interaction class example. At first a selection weight is assigned to each frequency available for selection ①. The selection weight is based on the following three parameters:

Average Tradeoff The tradeoffs of all previous lag executions with the corresponding frequency are averaged.

Sample History Size The selection weight considers how often a certain frequency has already been sampled for a given interaction class.

Sample History Trust Sample history trust indicates the accuracy of the calculated average tradeoff. It depends on the 95% confidence interval of the mean CPU busy time and the length of the sample history. The more samples that have been observed for this frequency and the smaller their confidence interval the higher the Trust factor. CPU busy time's confidence interval is chosen over lag duration since busy time scales almost linearly with CPU frequency. Animation timers and screen refresh intervals can lead to a minimum lag duration, which stops scaling with higher frequencies (see Chapter 4). Busy time, however, is unaffected by this problem.

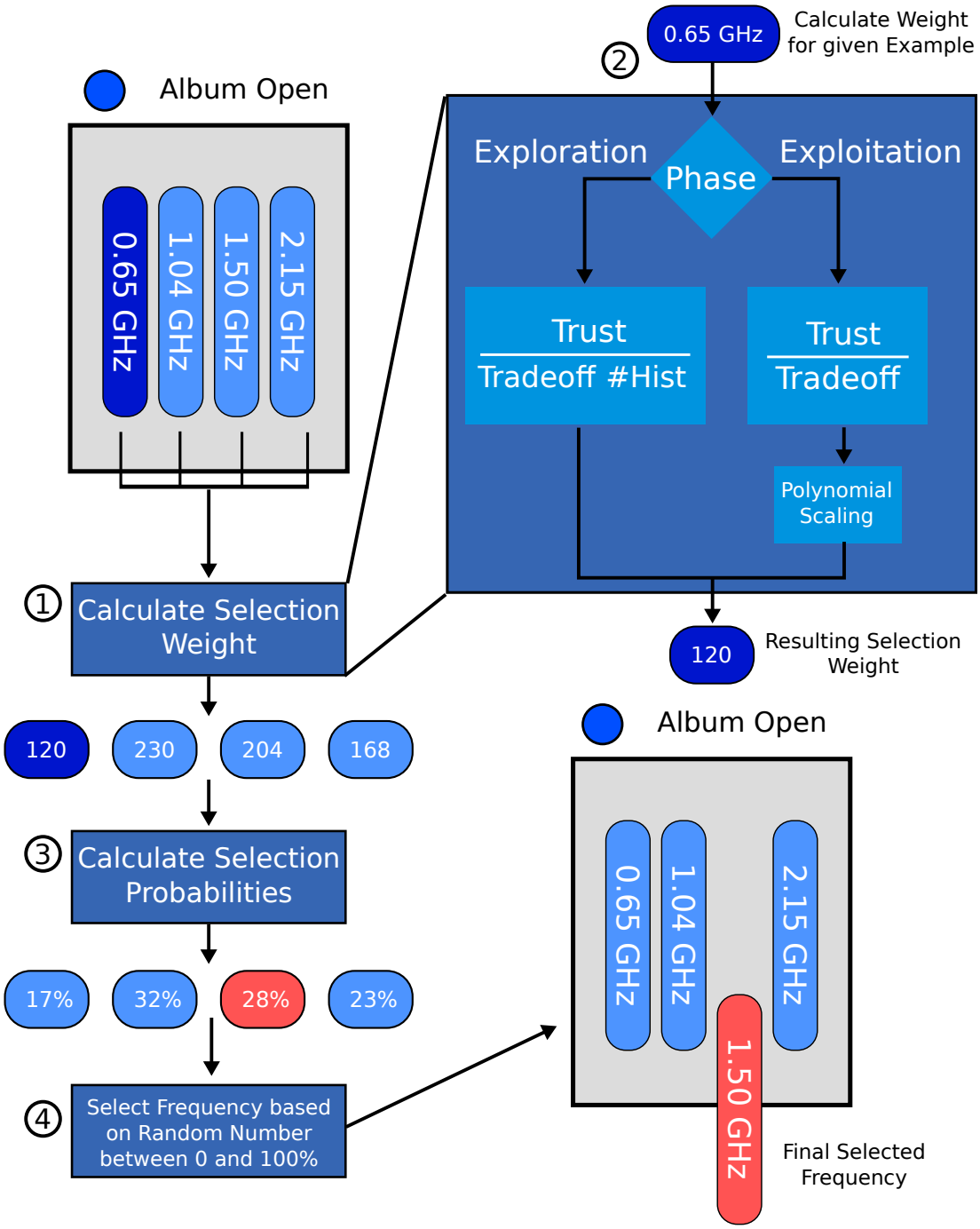


Figure 7.10: Frequency selection process for an upcoming interaction lag. ① Selection weights of all available frequencies are calculated ② depending on the frequency selection phase of the model. ③ Weights are translated to probability space and a ④ frequency selection is made by considering a random number between 0 and 100%. Frequencies with a higher selection probability are more likely to be chosen for the upcoming lag, their selection is, however, not guaranteed.

Depending on the current selection phase of the interaction class' behaviour model, those three parameters are combined differently to calculate the selection weight ②.

Exploration In the *Exploration Phase* the governor tries to get as good a picture as possible of the interaction class' behaviour. At the same time it still tries to maintain an acceptable user experience and energy consumption during lag execution. The selection weight of a single frequency is calculated by dividing corresponding trust by average tradeoff and history size. High trust and low tradeoff increase the selection weight of high tradeoff frequencies, while the history size adds additional weight to frequencies with fewer samples. This way *RLGov* carefully explores the frequency space whilst occasionally selecting frequencies which yield a high reward.

Exploitation In the *Exploitation Phase* the governor reduces exploration and puts a heavier weight on exploiting its collected knowledge. Therefore, the history size is not applied when calculating the selection weight. Here, trust is divided by tradeoff. In addition, the final selection weights of all frequencies are scaled with a polynomial function to increase selection probability for frequencies yielding high rewards even more.

After selection weights have been calculated, their values are translated into a probability space ③ so they add up to 100%. Based on the resulting probabilities, a final frequency choice is made by choosing a random number between 0 and 100% ④. Higher selection probabilities are more likely to be chosen as lag frequency, this is, however, not guaranteed. This selection policy causes an initial noisy period for instances of a single interaction class. Here, suboptimal frequency settings can be chosen. Once prediction accuracy exceeds a predefined threshold, the governor starts to settle on a frequency selection with minimal energy consumption and user irritation, i.e. minimal tradeoff. Now only sporadic samples of suboptimal selections are taken to account for behaviour changes. The next section will give more information on behaviour changes of interactions. The threshold configuration for the experiments conducted in this study is specified in Section 7.7.

7.6.2 Accounting for Behaviour Changes

By training a separate behaviour model for each newly encountered interaction class, *RLGov* is able to learn good frequency selections for unseen problems. This way it can keep up with new applications. It can also happen that an already known interaction class changes its behaviour and thereby the frequency choice with the lowest tradeoff. For example, an interaction starts a list search. Over time the list grows longer and the search algorithm needs more time to process it. This will lead to a constant increase in lag duration. With growing execution complexity the frequency with minimal tradeoff might change to a higher one to keep lag duration short and user irritation low. In this example behaviour changes slowly over time. Another example can be found when looking at application updates. An interaction lag requiring a high frequency could be updated by developers so it is suddenly able to execute sufficiently fast with a much lower one.

RLGov uses two mechanisms to handle behaviour change. Firstly, each history can be filled with samples up to a maximum amount (20 in the experiments for this study). As soon as a history is full and a new sample is added for the corresponding frequency, the oldest sample is discarded to make room for the new one. A sample history holding only the most recent entries allows *RLGov* to adapt behaviour. The second mechanism is due to the random sampling happening even if the *Exploitation Phase* is active for a model. *RLGov* will still occasionally keep looking at suboptimal selections to catch potential changes in application behaviour. Alternative approaches are certainly possible, such as flushing models for application updates or when encountering extreme differences. Exploring these possibilities is, however, left for future work.

7.6.3 Considering Missing Samples

In the *Exploration Phase* an additional technique is applied to avoid too frequent sampling of suboptimal frequencies. This technique estimates a reward, i.e. tradeoff, for yet unsampled frequencies. This allows *RLGov* to consider them for frequency selection without having actual execution data. The first time an interaction instance of an unknown interaction class is encountered, a default frequency is chosen (1.04 GHz for the experiments conducted in this study). For every future occurrence of interaction events (of the same class), execution statistics of the default frequency are scaled to get

an estimate for unsampled ones. The parameters that need to be scaled to calculate a selection weight and probability are tradeoff and trust. In the *Exploitation Phase* usually at least one sample for all frequencies is present and scaling is no longer required. The following section will explain the scaling process in more detail.

7.6.3.1 Scaled Tradeoff

Figure 7.11 shows details of the scaling process for the Album-Open interaction. The corresponding behaviour model is in an example training state where all frequencies have been sampled except 1.50 GHz. To scale the default frequency's tradeoff, a scale factor is needed. This scale factor is calculated by considering the history entries of all frequencies sampled so far. Default tradeoff is not scaled directly, instead the default frequencies average CPU busy time and lag duration are scaled. Using those parameters, the missing tradeoff value is then calculated the regular way. The scale factor between default frequency and missing frequency is interpolated using scale factors between the default and all other frequencies.

In detail: In a first step ① a scale factor between the default frequency's average CPU busy time and the average busy times of each other frequency is calculated. The default frequency (1.04 GHz) is chosen as the baseline because it definitely has at least one sample entry. For sampled frequencies the average CPU busy time is divided by the average default frequency busy time. For unsampled frequencies, such as 1.50 GHz in the example, no average busy time is available yet. Therefore, the values of the frequencies (in GHz) are divided. Here the default frequency's value is divided by the unsampled frequency's value.

Once a busy time scale factor for each frequency is calculated, polynomial regression is used to interpolate the final scale factor ②. A quadratic curve is fitted to the generated scale factors. The intention is to smooth out inaccuracies from using frequency GHz value scale factors for unsampled frequencies by compensating with CPU busy time scale factors of neighbouring sampled ones. The final scale factor is taken from the fitted curve and used to scale the average tradeoff from the default frequency ③. Specifically, average CPU busy time and average lag duration of the baseline are scaled with the determined factor and the results are used to calculate energy consumption and irritation and eventually the scaled Tradeoff (see Section 7.5).

The algorithm to calculate scaled tradeoff for an unsampled frequency is shown in

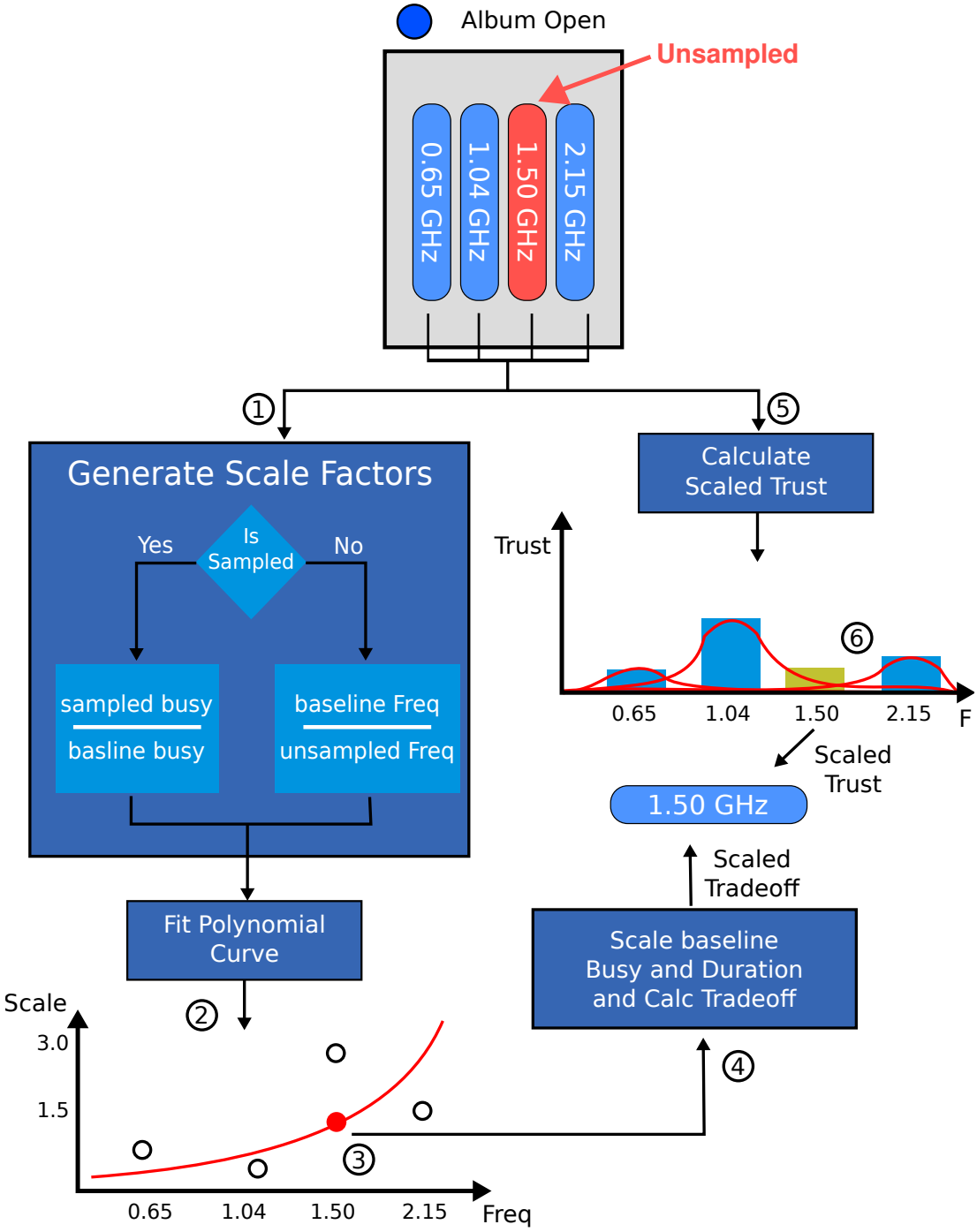


Figure 7.11: Process to scale tradeoff and trust for an unsampled frequency using the default frequency as the baseline. Scaled tradeoff is calculated using polynomial regression over scale factors of sampled frequencies. Scaled trust is calculated by blurring the trust of neighbouring sampled frequencies.

```
1  function scaleTradeoff(defaultFreqStats, unsampledFreq) {
2    freqScaleX = list()
3    freqScaleY = list()
4    for each available frequency f {
5      frequencyStatistics = getFreqStats(f)
6
7      freqScaleX.append(f)
8      if(frequencyStatistics.wasSampled()) {
9        freqScaleY.append(
10         frequencyStatistics.mean_busytime /
11         defaultFreqStats.mean_busytime)
12     } else {
13       freqScaleY.append(defaultFreqStats.frequency / f)
14     }
15   }
16   freqScaleModel = fitQuadraticCurve(freqScaleX, freqScaleY)
17   return freqScaleModel.predictScaleFactor(unsampledFreq)
18 }
```

Figure 7.12: An interpolated scale tradeoff factor is calculated by fitting a polynomial model to scale factors of sampled and unsampled frequencies.

pseudo code in Figure 7.12. In line 4, a loop iterates over all available frequencies and calculates scale factors which are used to train the regression model in line 16. A quadratic curve is fitted to calculated x and y values. x values are the corresponding frequencies while y values are calculated scale factors. Using the fitted curve, an interpolated scale factor is returned for the unsampled frequency in line 17. Within the loop frequency busy time is divided by default frequency busy time, if the frequency was already sampled. If not, the GHz frequency values are divided.

Figure 7.13 shows an example of a scale factor curve for an actual experiment. The corresponding interaction class has been sampled 20 times and 9 out of the 14 frequencies have been seen. The missing frequencies are 0.42, 0.65, 0.88, 0.96 and 1.96 GHz. Each point stands for a calculated scale factor either by using CPU busy time for sampled profiles (circle) or frequency values for missing ones (triangle). The dashed line shows the fitted curve for the given data points. The ribbon around it indicates its 99% confidence interval. The solid line shows actually observed results for scale factors after several hundred samples of corresponding interaction instances were executed and all frequencies have been observed multiple times. For most missing frequencies, the scale factor interpolation curve corrects the estimated value in the right direction. Hence, this method is useful when estimating a tradeoff for unknown frequencies.

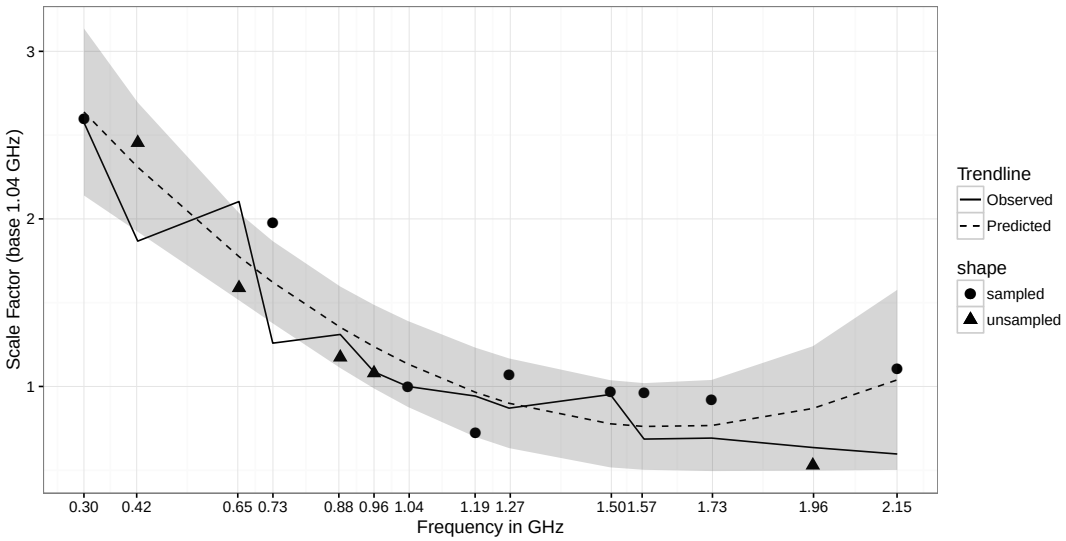


Figure 7.13: A frequency scale factor model for an example interaction. Each dot represents a calculated scale factor for sampled frequencies, each triangle a scale factor for an unsampled ones. The dashed line shows the fitted estimation curve and the solid line shows real observed values after all frequencies have been sampled to a high degree.

7.6.3.2 Scaled Trust

```
1  function calcScaleTrust(unsampledFreq) {
2    trustFactor = 0;
3
4    for each available frequency f {
5      frequencyStatistics = getFreqStats(f)
6      if(frequencyStatistics.wasSampled()) {
7
8      freqDistance = absolute_value(f − unsampledFreq);
9      trustFactor +=
10     (frequencyStatistics.trustFactor ∗ NEIGHBOURING_TRUST_SCALE)
11     / (NEIGHBOURING_TRUST_DECREASE_FACTOR ∗ freqDist);
12   }
13
14   return trustFactor;
15 }
```

Figure 7.14: A trust factor for an unsampled frequency is calculated by considering partial trust values of neighbouring frequencies.

Since selection probability is generated by considering tradeoff and trust, the later parameter needs to be scaled for the missing frequency too. This process is depicted on the right of Figure 7.11 ⑤. Understandably, the trust for a scaled frequency cannot be high. The scale factor to calculate tradeoff is generated by considering execution

statistics of neighbouring frequencies that have already been sampled. Therefore, the better known the neighbouring frequencies are and the higher their trust, the higher the scaled trust of the missing one. Each trust value of a sampled frequency is multiplied by a decay factor depending on its distance to the missing frequency. The results are added up and equate to the final scaled trust ⑥. Now all statistics are present for the missing frequency to be included in the selection probability calculation. The following section will show examples for how described techniques help *RLGov* to settle on a frequency with low energy and irritation.

Figure 7.14 shows a pseudo code algorithm for calculating the scaled trust of an as-yet unsampled frequency. A for loop iterates over all sampled frequencies and calculates a distance value between the current sampled frequency and the unsampled one in line 8. The trust factor is accumulated by considering trust from each other sampled frequency. In line 9 - 11 the partial trust for the current frequency of the loop is calculated. The trust factor of the current sampled frequency is weighted (0.25 in the experiments for this chapter) and divided by the trust decrease over frequency distance. The further the frequencies are apart, the less trustworthy their previously conducted scaled tradeoff.

7.6.4 Frequency Settling

Figure 7.15 shows frequency prediction results for a range of example interaction classes from the workload used in this study. The selected frequency is shown for each sample of an instance of the corresponding interaction class over the course of the workload execution:

Class A Interaction A shows a clear difference between *Exploration Phase* and *Exploitation Phase*. The phase switch happens after 66 samples. Around that time the frequency settles at 1.5 GHz.

Class B *Exploration Phase* of interaction B is over after 39 samples, however, during the *Exploitation Phase* the selection still alternates among frequencies between 2.15 and 1.5 GHz before it settles at 1.96 GHz. This is due to noise in the data where more data samples keep increasing the model's accuracy during *Exploitation Phase*.

Class C Exploration of interaction C ends after 21 samples but the frequency never

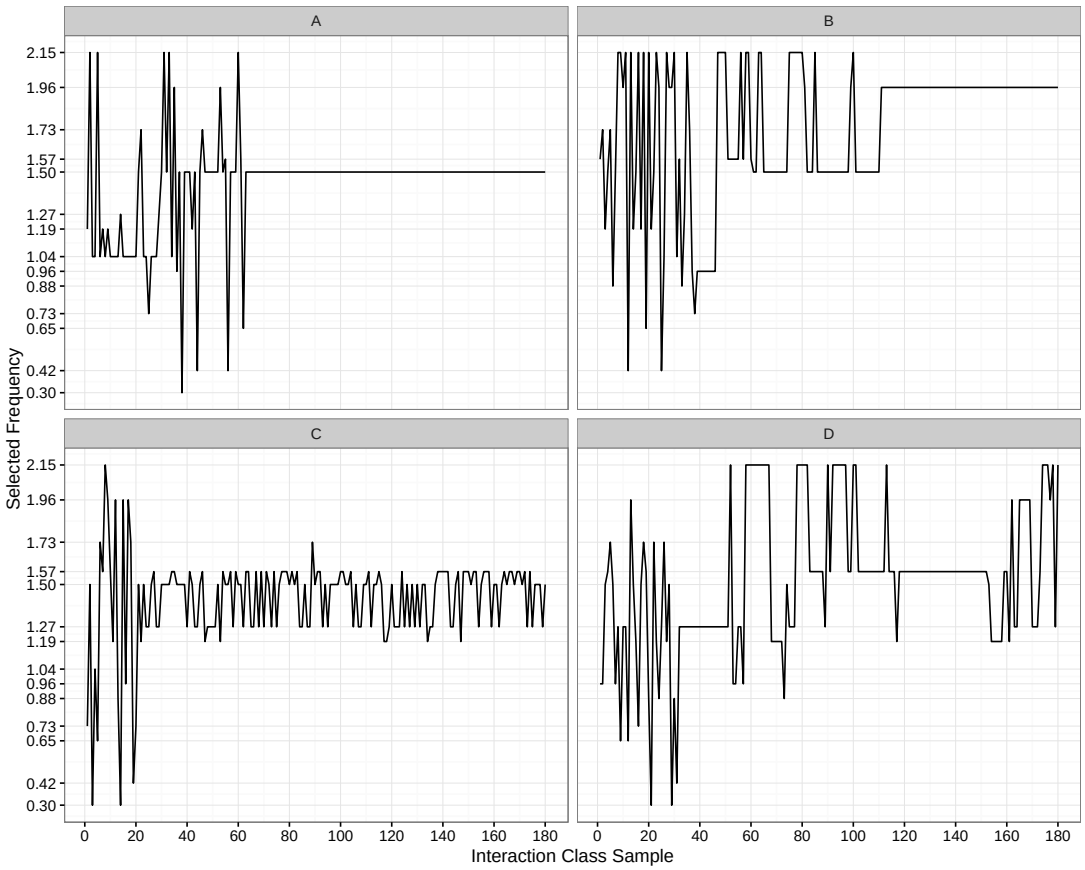


Figure 7.15: Selected frequencies for multiple instance samples of example interaction classes. Class A and B settle clearly on a single frequency. Class C settles on a range of frequencies with similar tradeoffs and Class D never clearly settles at all.

seems to clearly settle on a single one. It keeps alternating in the frequency range between 1.57 GHz and 1.19 GHz. This is due to frequency tradeoff values being close together even after many samples have been observed. This results in probability values being similarly high within the range of alternating frequencies. An additional third selection phase could improve this behaviour where the lowest tradeoff frequency is forced to be taken after a second model accuracy threshold has been reached.

Class D Even though the *Exploration Phase* of interaction D is complete after 32 samples, and selected frequencies shift towards the upper half of the frequency spectrum, a settling never happens. In this case the cause is due to the same interaction identifier being assigned to interaction events with two or more different execution behaviours (see Section 7.3). To solve this a modified way of identifying interaction events needs to be applied.

The frequency selection algorithm presented in this section allows *RLGov* to train behaviour models for encountered interaction classes. During exploration the governor tries to keep the user experience and energy consumption on an acceptable level by focusing on exploring frequencies considered best so far. Additionally, the tradeoff for unsampled frequencies is estimated by scaling sampled frequency statistics. Once the frequency space is searched to a certain degree *RLGov* switches to the *Exploitation Phase* and heavily weights frequencies yielding a low tradeoff. Now, for most interaction classes the selected frequency settles on a single one or within a small range where tradeoff values are close together.

7.7 Experimental Setup

In the following sections the results of a series of experiments will be presented. *RLGov* was executed and evaluated for the benchmark workload generated in Chapter 4. Evaluation focus is on the overall energy consumption and user irritation compared to three standard frequency governors which are currently used on most *Android* mobile devices. Namely, those are the *Conservative*, *Interactive* and *Ondemand* governors. Additionally, the results will be put in the context of the baseline frequency profile generated in Chapter 5 by showing how close *RLGov*'s results are to the *Oracle*'s.

The workload executed in this study is the same as the one used for the experiment in the previous three chapters. It is composed of 16 datasets from different users with a length of about 10 to 15 minutes each. It has a total length of 190 minutes with 1935 user input events, i.e. interaction instances. As in previous experiments, the leading and trailing interaction instances of each dataset were omitted. They are used to activate and deactivate workload recording and are therefore not part of the actual workload. Also, some interactions within the workload were omitted since the recording method presented in Chapter 4 could not handle them (see detailed workload description in Section 4.4). This leaves 1852 interaction instances for the analysis.

The irritation model's Gompertz function parameters chosen for the experiments conducted in this study are roughly: $a = 150$, $b = 5$ and $c = 0.83$. They cause irritation to settle on a maximum after about 10 seconds. The tradeoff is calculated from energy and irritation using: $T = E \times I^2$.

The experiments conducted here focus on analysing *RLGov*'s functionality rather than

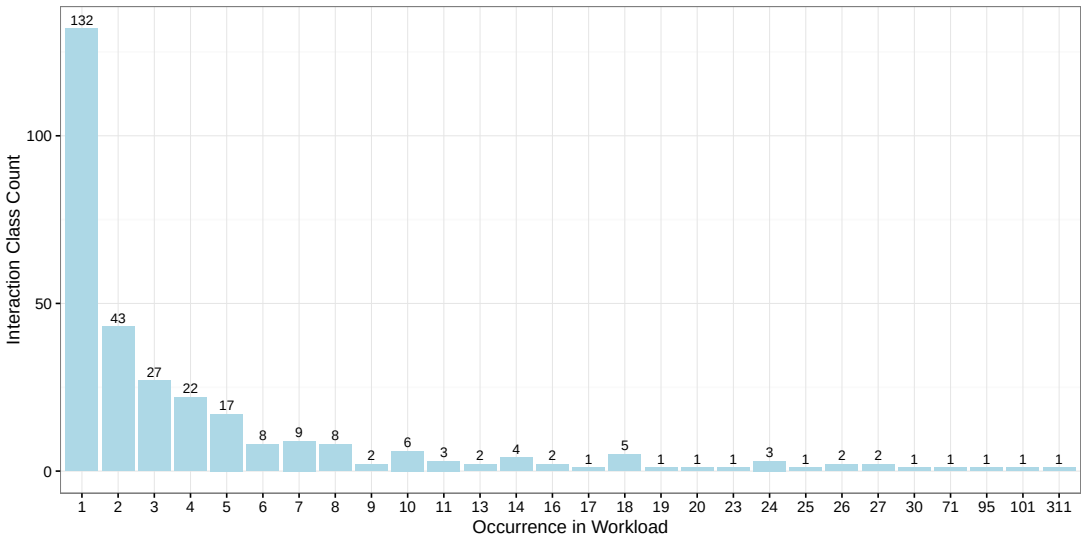


Figure 7.16: Distribution of interaction class encountered across the complete workload. The x-axis shows how often a class was encountered while the y-axis shows how many classes appeared with that encounter rate.

analysing how well interaction instances can be identified at runtime. Therefore, interaction ids were given to the single interaction events in the workload manually considering the mechanics described in Section 7.3. In total 307 interaction classes were identified.

Figure 7.16 shows the distribution of interaction classes in the workload. The x-axis indicates how often an interaction class is encountered and the y-axis shows a count of interaction classes. For example, 132 interaction classes are encountered once, 5 classes 17 times and a single class 311 times. The interaction classes appearing a single time within the workload make up 43% of all classes. That means that a single interaction instance of those classes is executed during the workload. Among them are opening the article options in the Pulse news application, creating a new contact in the phone book or opening the city map in the Stay application. This number is particularly high due to the somewhat artificial setting in which the workloads were generated (see Section 4.4). Users were asked to exercise a foreign phone for a block of 10 minutes in contrast to using their own in the way the would during their daily routines.

However, in a more realistic setting there would also be interaction classes which the mobile device user only executes one or two times. This can happen, for example, when the user did not like an application and removed it right away. As described in Section 7.6, *RLGov* chooses a medium frequency by default for unseen interaction

classes which mostly produces acceptable results in terms of user irritation and energy consumption. For lags which appear more often so that the user cares about optimal performance, *RLGov* has the chance to learn from multiple instances and optimise its behaviour after a warmup phase. Examples are interaction classes with many instances in the workload such as typing a key on the on-screen keyboard or opening a picture in the gallery.

For future work *RLGov* could be evaluated on user devices while they go about their daily routines. This will produce longer workloads with more interactions. However, for the initial development performed in this study the recorded workloads are much better suited due to quick turnarounds between developing mechanics and evaluating them for a relatively short workload. To get a better picture of how *RLGov* handles frequent interactions, the entire recorded workload is executed multiple times during the experiments. Over the course of all executions the same interaction class models are maintained and updated. This way all interaction classes occur multiple times and *RLGov*'s learning algorithm can be evaluated over a larger data set.

For a single benchmark run, the workload is chained together 200 times while maintaining the same interaction class models. The chained workload will be called the *extended workload*. It has a length of about 633 hours or 26 days pure interaction time. Considering a mobile phone usage time of 174 minutes over a 24 hour period [179], the extended workload represents 218 days of phone usage. To get statistically sound data and calculate error margins, execution of the extended workload is repeated 5 times. The standard error is calculated across those 5 iterations and displayed as error bars on graphs. To reduce execution time and provide a flexible development environment the workload is executed in a simulator. For that purpose the interactive workload simulator used in Chapter 6 is extended with the *RLGov* algorithm. Specifics are described in the next section. It is able to run all 5 iterations in about 27 hours or about a day on an Intel Xeon E5-1620 processor. Compared to real time execution on a mobile platform of 3167 hours or about 131 days, this is roughly a 118$\times$ speedup.

7.7.1 Simulator *RLGov* Extension

The mobile workload simulator used to evaluated the lag end detection heuristic in the previous chapter (see Section 6.4.1) is extended with a frequency predictor. With that it is able to simulate the frequency selection process for a given interactive workload

generated in Chapter 4 with speedups around 100×. The workflow of the extended version is depicted in Figure 7.17.

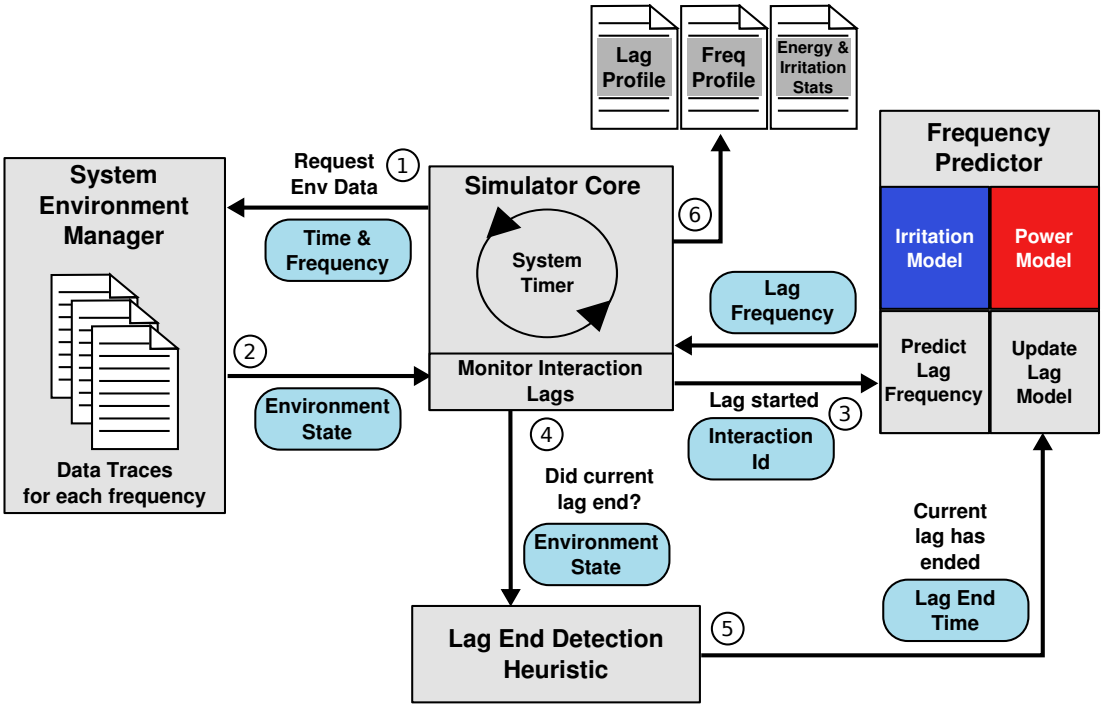


Figure 7.17: ① The simulator core requests the environmental state for the current time and CPU frequency. ② The System Environment Manger picks the correct data trace and returns it. ③ When a lag start is detected, a corresponding interaction id is passed to the frequency predictor which predicts a lag frequency. ④ While the lag is active LDH is polled for the lag end. ⑤ As soon as it is detected, the measured lag duration is forwarded to the Frequency Predictor and used together with the Irritation and Power model to update the interaction classes' QOE model. ⑥ The CPU is set to idle period frequency. At the end simulation statistics on lags, selected frequencies and energy and irritation data are reported.

As done before, the simulator executes a timer which updates the simulation time with millisecond accuracy. At the start of each simulation loop, the current state of the system environment is requested from the System Environment Manager. The Simulation Core now does not only provide a timestamp but also the currently active CPU frequency ①. The environment manager preloads workload system traces for all available CPU frequencies and can provide the corresponding system state at that time for the given frequency ②. A system state contains information on whether the CPU is currently busy or idle, if a screen refresh has happened and so forth.

As soon as the beginning of a lag was noted by observing user input, the Frequency Predictor is notified and a corresponding interaction id is provided ③. It predicts a

good CPU frequency for the lag based on the current training state of the interaction classes' QOE model. The LDH is now polled regularly for the end of the active lag ④. Once it is discovered, the measured lag duration is passed on to the Frequency Predictor ⑤. With that lag ending and other necessary lag execution statistics (see Section 7.5) energy and user irritation are calculated using the internal models. The resulting tradeoff is used to update the corresponding QOE model and the frequency prediction accuracy is adapted. Afterwards, the CPU is fixed to the default idle frequency until the beginning of the next lag is detected. Eventually, log files for a lag profile, a frequency profile and energy and irritation statistics are generated.

7.8 Experimental Results

This section will present results from experiments conducted with the user perception based learning governor developed in this chapter. It will show an evaluation of frequency predictions in the first part and overall energy consumption and user irritation in the second.

7.8.1 Frequency Prediction Results

RLGov goes through an initial *Exploration Phase* when a new interaction class is encountered. During that phase semi-random frequency samples are taken to explore lag behaviour across the frequency spectrum. *RLGov* tries to keep suboptimal frequency settings as infrequent as possible and to quickly find the best fitting one. Since multiple instances of the complete workload are chained together while maintaining the same behaviour models, *RLGov*'s optimisations can be evaluated for all interaction classes in the workload.

7.8.1.1 Frequency Space Exploration

Figure 7.18 shows how *RLGov* explores the frequency space for interaction classes appearing in the workload. The complete frequency space consists of 14 different frequency points ranging from 0.3 GHz up to 2.15 GHz. Those are the possible frequency settings of the Snapdragon 800 processor used for workload recording in Chapter 4. The entire frequency space of a single interaction class is searched as soon as each of

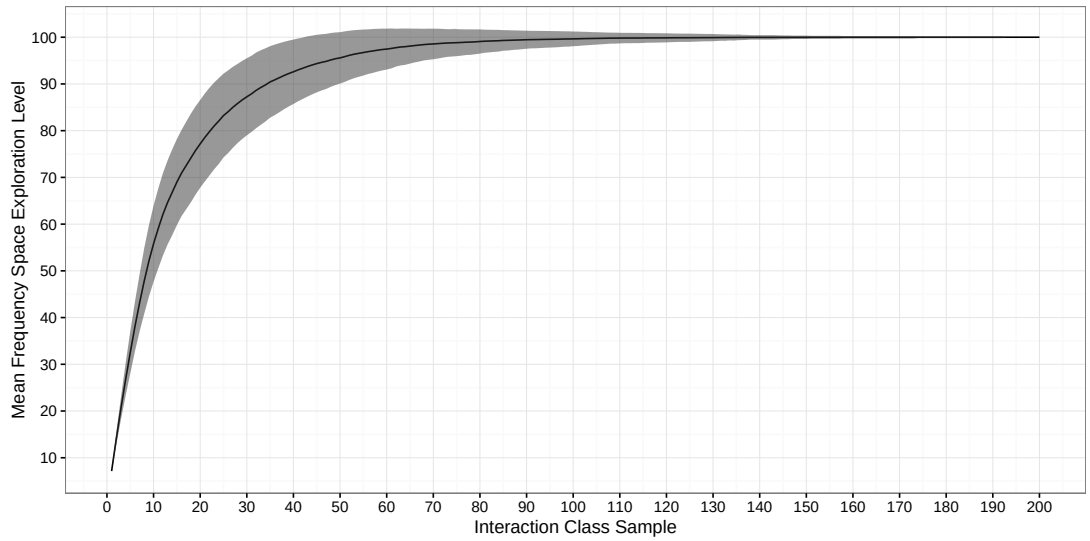


Figure 7.18: Mean frequency space exploration level of all interaction classes over class samples during workload execution.

the 14 frequencies was sampled at least once for that class. On the y-axis is a mean of the frequency space exploration level over all interaction classes in the workload. The x-axis shows how often an interaction class was sampled by *RLGov*. The ribbon around the graph shows the standard error calculated over all interaction classes within the 5 execution iterations of the extended workload.

The mean level of frequency space exploration has a standard deviation of zero after each interaction class has been sampled 192 times. That means at this point the entire frequency space of each interaction class in the workload has been sampled. The largest part of interaction classes, however, has been completely sampled much sooner. The minimum amount of samples needed to to search the whole frequency space of a single class would be 14 samples, since this is the number of available frequencies. After each class has been sampled at least 14 times, on average 70% of all available frequencies have been observed for each class. This value grows to 97% after 60 samples.

In Section 7.6 it is described that *RLGov* semi-randomly samples the frequency space for a single interaction class to learn its behaviour. This is done in two different phases: In the initial *Exploration Phase RLGov* puts a slight weight on unsampled frequencies. In the *Exploitation Phase* a heavy weight is put on the frequency with the best energy and irritation tradeoff. The change between the two phases happens for each interaction class separately and is done as soon as the frequency space exploration level of the class

goes above a threshold. In the experiments conducted in this study, this threshold is set to 100% to make sure the entire frequency space is covered. Even though *RLGov* searches the whole frequency space it tries to keep the amount of suboptimal samples low. This is why the frequency space is not completely searched for all classes within the minimum possible time of 14 samples. Knowing the whole space is, however, not necessary to find good frequencies for interaction lags. The next section will show that in many cases, the minimum tradeoff frequency for an interaction class was found before the entire frequency space was sampled.

7.8.1.2 Best Tradeoff Frequency

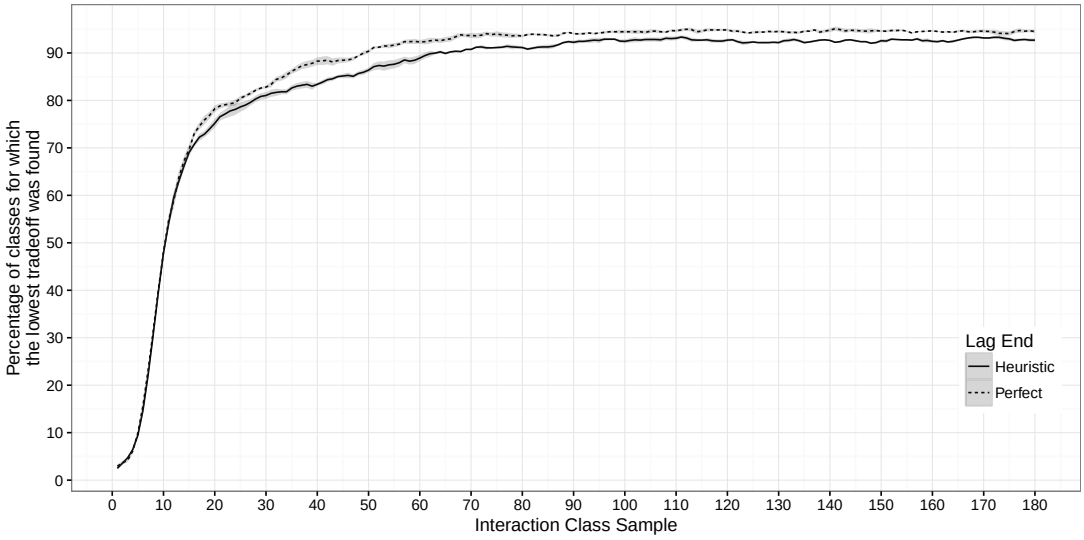


Figure 7.19: The y-axis shows percentage of interaction classes for which *RLGov* has identified the lowest tradeoff frequency. X-axis shows how often an interaction class has been sampled.

Figure 7.19 shows after how many samples *RLGov* identified the lowest tradeoff frequency for all interaction classes in the workload. Again, the x-axis lists interaction class samples over the course of the workload. The y-axis reports the percentage of interaction classes for which the lowest tradeoff frequency was found. A ribbon is used to show the standard error over all interaction classes within the 5 executions of the extended workload. However, the error is small and therefore barely visible.

Using its internal power and irritation models, *RLGov* is able to identify the frequency for each interaction class which shows the lowest tradeoff. However, this is not necessarily the frequency identified in the *Oracle* study as the optimal frequency for the lag.

As described in Section 5.4 the *Oracle* frequency profile is calculated for each interaction lag, e.g. interaction instance, in the workload. However, *RLGov* is optimising the frequency for each interaction class which can comprise multiple interaction instances (see Section 7.3).

Lowest Tradeoff Found Metric The graph in Figure 7.19 uses a metric to indicate after how many samples of a single interaction class *RLGov* could identify its lowest tradeoff frequency. This metric applies a sliding window calculation over all samples of a single class. For each sample executed of a class its corresponding behaviour model is searched for the current frequency with the lowest average tradeoff, namely the current lowest tradeoff frequency. If an interaction class' current lowest tradeoff frequency has a coefficient of variation below 10% over a window of 20 samples, the lowest overall tradeoff frequency for that class is considered to be found. The window has a length of 20 since this is the maximum length of a frequency's sample history (see Section 7.6.2).

The solid line in the figure shows the results for a workload execution using the LDH described in Chapter 6. The dashed line shows the result for an execution where *RLGov* has perfect knowledge of the real lag ending as measured with the video markup method from Chapter 4. The difference between them is only a few percent at most. These results show that *RLGov* is able to compensate for most of the detection errors made by LDH (see Section 6.5). Overdue lag end detection for very low frequencies is compensated because very low frequencies usually do not have good tradeoff values even if lag end detection would be perfect for those cases. When the detected end lies after the real lag, which is the case for most lags, the resulting user irritation is not affected. As was discussed in Section 4.2, when the user perceives a lag ending he does not care about performance during the following idle phase. Keeping the lag frequency active for longer than necessary, therefore, only affects the energy consumption. However, the resulting surplus on energy consumption is rather low when the detection delay is not too long because the idle period is usually low on CPU intensive work. The only detection errors which have a noticeable negative impact are highly premature end detections. They lead to much shorter lag durations than in reality and drastically reduce resulting lag energy consumption and user irritation. The resulting tradeoff is therefore good and would be favoured.

After sampling each class 14 times, *RLGov* was able to find the lowest tradeoff frequency for 66% of them. This value grows to 89% after sampling each class 60 times.

After 200 workload iterations, *RLGov* was able to identify the lowest tradeoff frequency for 93% of all interaction classes. This value is reached after 90 samples of each class and remains on that height. The graph never fully reaches 100% because of noise between interaction instance of the same class. A larger data sample can help to further investigate this behaviour. These results show, that *RLGov* is able to find the frequency with the lowest energy and irritation tradeoff for most interaction classes after an initial sampling period. They also show that LDH lag end predictions are accurate enough to achieve good results.

7.8.2 Overall Energy and Irritation Results

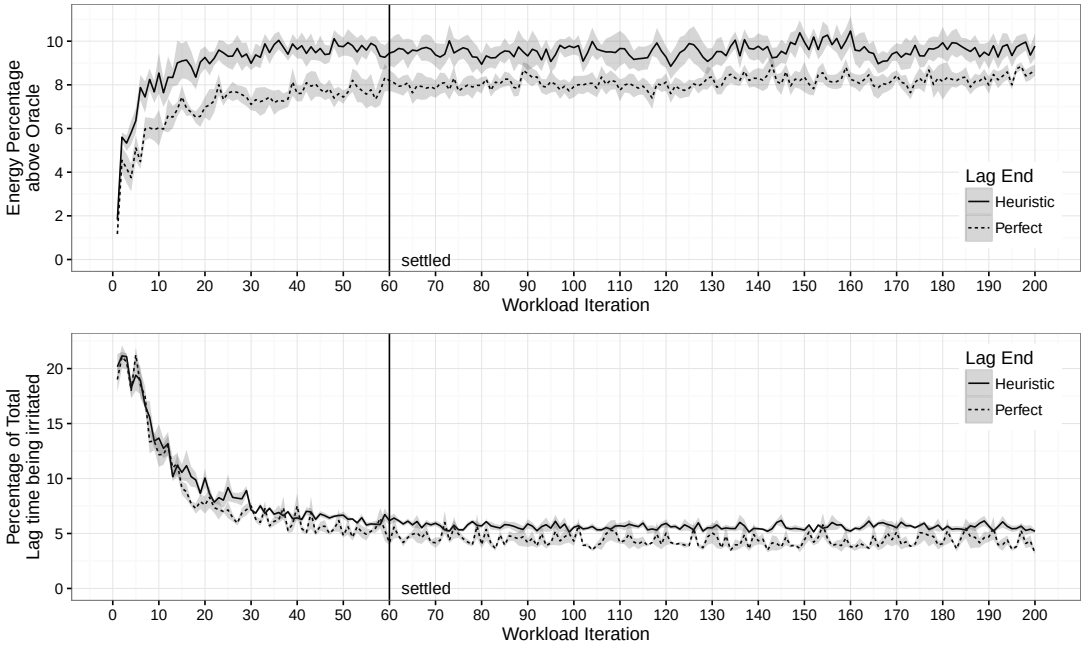


Figure 7.20: This graph shows total energy consumption and user irritation development over the course of the extended workload. The y-axes of both graphs show energy percent above *Oracle* results and the total percentage of lag time the user is irritated. The x-axes show how often a single workload was executed. They sum to the 200 iterations of the extended workload. A vertical line marks where most of the interaction classes settled on a lowest frequency.

Figure 7.20 shows how total energy consumption and total user irritation develop over the course of the 200 workload executions of the extended workload (see Section 7.7). A single workload iteration consists of an execution of each of the 16 recorded datasets. The generated interaction class models are maintained over the course of all workload iterations and are continuously updated with sample data. The ribbons around the

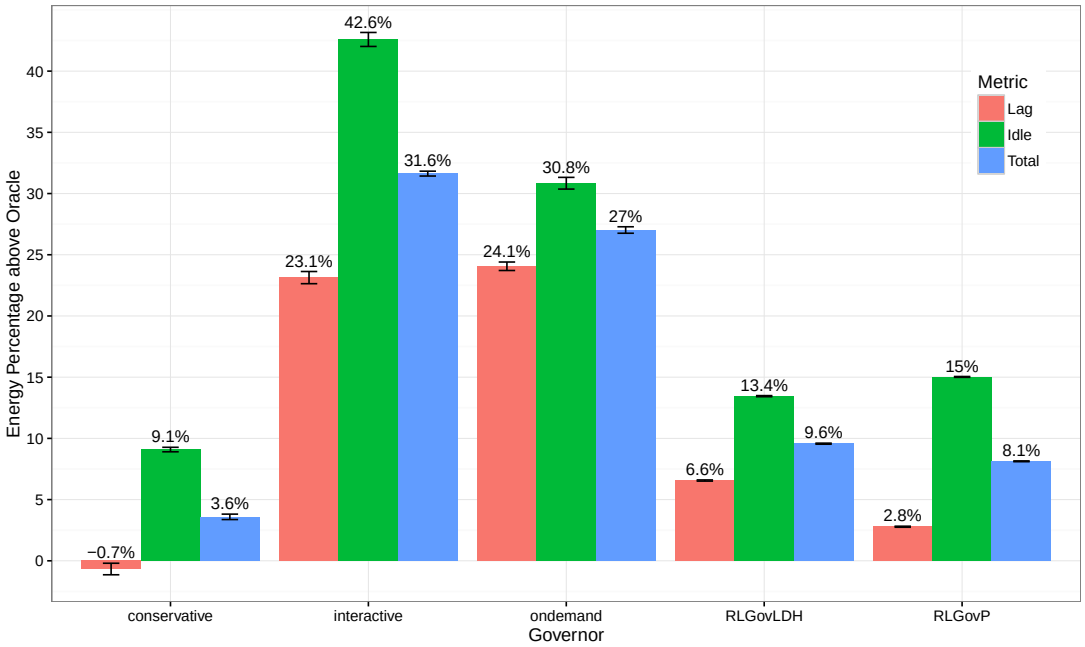
graphs show the standard error calculated over 5 executions of the entire extended workload. As in Figure 7.19, the solid line shows results of *RLGov* using the LDH while the dashed line uses perfect knowledge of real lag endings. The top graph shows by what percentage *RLGov*'s total energy consumption exceeds the *Oracle*'s frequency profile from Chapter 5. The bottom graph shows the total percentage of time the user is irritated during interaction lag execution.

Per definition the *Oracle*'s approach has a user irritation of zero. The same user irritation metric as for the experiments in Chapter 5 is used for the evaluation of *RLGov*'s results. The irritation for each interaction instance in the workload (independent of its corresponding class) is based on the execution results of the fastest possible frequency. For the fastest frequency, the irritation is zero. If a different frequency exceeds the lag duration the fastest frequency could achieve including a small additional margin, the overflow time is considered as user irritation value for that lag (see Section 5.2 for details). The irritation metric is calculated offline after *RLGov* finished workload execution and lag durations are known.

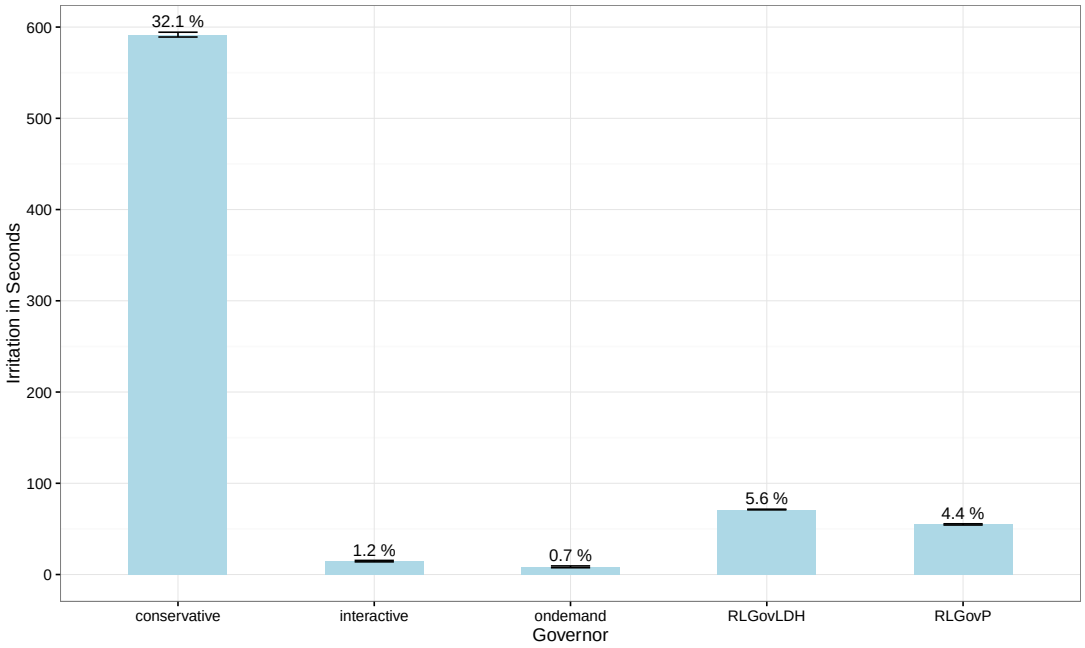
The total energy consumption begins close to the *Oracle*'s and quickly grows up to approximately 10% above it, where it settles. *RLGov* using perfect knowledge of lag endings achieves slightly better results than *RLGov* using the heuristic. The difference is approximately 3%. Total irritation starts of around 20% and settles at approximately 6% after about 60 workload iterations. Again the governor version using perfect knowledge of lag endings is only approximately 2% better. The vertical lines in Figure 7.20 labelled with "settled" mark the workload iteration at which point most interaction classes have settled on a lowest frequency. To summarise these results, average energy consumption and user irritation over all workload iterations following the 60-iteration mark are considered in the next graphs.

Figure 7.21a shows a comparison of *RLGov*'s energy results after most interaction classes settled on a minimum tradeoff frequency. They are compared to the three standard governors also considered in Chapter 5. The energy consumption is displayed in percent above what the *Oracle*'s frequency profile could achieve. Next to total, energy consumption results are shown for lag and idle periods alone. Again, *RLGov* was executed twice: once using the LDH (RLGovLDH) and once using perfect lag end knowledge (RLGovP).

None of the governors reach the energy consumption of the *Oracle*'s frequency pro-



(a) Comparison of total energy consumption above the *Oracle* profile between the three standard governors and *RLGov* using the LDH (RLGovLDH) and *RLGov* using perfect lag end knowledge (RLGovP). Next to a total, energy consumption is also displayed for lag and idle periods alone.



(b) Comparison of total user irritation between the three standard governors and *RLGov* using the LDH (RLGovLDH) and *RLGov* using perfect lag end knowledge (RLGovP). Each bar is annotated with the percentage of time the user is irritated during interaction lag execution.

Figure 7.21: These graphs shows a comparison of the energy consumption and user irritation for all three standard governors and *RLGov* after it settled for most of the interaction classes in the workload.

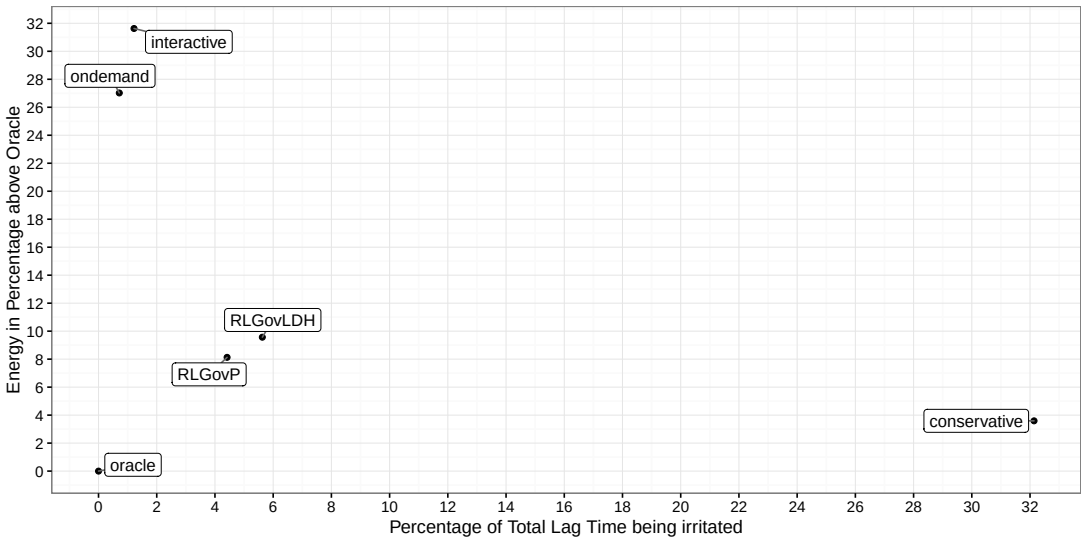


Figure 7.22: This figure shows energy consumption and user irritation for all standard governors and *RLGov* using the LDH (RLGovLDH) and perfect lag end knowledge (RLGovP). Energy and irritation results are displayed in relation to the *Oracle* results. The *Oracle* lies at the origin of the graph.

file. Closest is *Conservative* with 3.6% followed by *RLGov* with 9.6%. Using perfect lag ending knowledge *RLGov* manages to get 1.5% closer to the *Oracle*. *Interactive* and *Ondemand* need 31.6% and 27% more energy. For all five governor runs the energy difference to the *Oracle*'s profile is highest during idle time. During lag time *Conservative* achieves with -0.7% even slightly better results than the *Oracle*. This is, however, done by trading in a significant amount of user irritation as will be shown in Figure 7.21b.

In Figure 7.21b the governors' accumulated user irritation over the course of a single workload execution is shown. The time value displayed on the y-axis can be understood as a total time for which the user was irritated during the execution of the workload. The percentage above each bar shows which percentage of the total irritation lag time for each governor that was. *RLGov*'s user irritation is with 5.6% slightly higher than *Interactive* and *Ondemand* with each about 1%. It is, however still far below *Conservative* with 32%. *RLGov* using perfect lag end knowledge is only 1.2% better than *RLGov* using the LDH. This shows that the LDH's detection error compared to the real lag endings does not have a major impact on the final result.

Figure 7.22 shows all governors for a combination of both energy consumption in percent above the *Oracle* and the percentage of time the user is irritated during lags. *RLGov* is able to outperform *Interactive* and *Ondemand* in terms of energy consumption.

It is able to save up to 22% energy compared to the two governors. *Conservative* still needs 6% less energy than *RLGov*, it is, however, much more irritating to the user. The QOE metric shows that *RLGov*'s user irritation during lag execution is 26.5% better than *Conservative* and only 5.6% above the all knowing *Oracle*. Several optimisation strategies can be applied to make *RLGov* match the *Oracle* closer:

1. Frequency selections are optimised for interaction classes and not single interaction instances as performed by the *Oracle*. The noise of single instances belonging to the same class can cause the lowest tradeoff frequency to shift with each sample. This is especially true if instances have been put in the same class which don't show the same behaviour. An improved instance identification method can help to fix this problem.
2. The *Oracle* optimises frequencies for user perceived idle periods while *RLGov* always chooses the same.
3. The runtime irritation model used by *RLGov* deviates from the offline metric used to evaluate final results. Improving the online model to better match the offline QOE metric can help *RLGov* to better learn frequency behaviour.
4. A longer workload containing more actual samples of interaction instances can give a better picture of training results. The method presented in Chapter 4 can be used to generate them.
5. Even though differences in results between the LDH and *RLGov* using perfect lag end knowledge were small, an improved heuristic can be beneficial.

7.9 Conclusion

In this chapter a runtime solution was developed to improve DVFS energy efficiency for interactive mobile workloads whilst providing a good QOE to the end user. By using a heuristic to distinguish between interaction lag and idle periods, the implemented governor is able to consider the user's point of view. A reinforcement learning based approach helps *RLGov* to learn interaction lag behaviour for different frequency settings and allows it to find lag frequencies with low energy consumption and user irritation for up 93% of the evaluated cases. In so doing, it could successfully exploit the energy saving potential identified in Chapter 5 and closely match an all knowing

Oracle study. Overall, *RLGov* was able to achieve improved results compared to the current standard frequency governors on *Android* mobile devices. It was able to beat *Interactive* and *Ondemand* in terms of energy consumption and *Conservative* in terms of user irritation. Inaccuracies while detecting the lag ending were causing only minimal differences compared to using perfect knowledge of the lag ending as perceived by the user.

Chapter 8

Conclusions

8.1 Introduction

The research goal of this thesis was to improve the energy efficiency of current DVFS algorithms for interactive mobile workloads. Presented methods and experiments were motivated by the observation that current DVFS techniques work well for batch workloads but leave room for improving energy consumption when it comes to interactivity. To provide satisfying response times to user interactions, current governors waste energy by setting higher performance levels than necessary. Additionally, they raise the CPU clock speed when the user is imperceptible or indifferent to performance changes. Based on these observations, it is claimed that information on how the end user perceives workload performance can be exploited to improve DVFS energy efficiency whilst maintaining high QOE. Such information helps to learn how frequency settings affect user perceived performance, which will eventually allow finding optimal settings.

The following Section 8.2 will summarise the methods, tools and experimental results that were presented in this thesis to find supporting evidence for the initial research claim. The summary is followed by a critical analysis of conducted research in Section 8.3 and a description of future work in Section 8.4.

8.2 Summary

This thesis presented four studies on how to improve DVFS energy efficiency using information on user perceived performance levels. The first described a methodology to benchmark QOE for an interactive mobile workload. No current mobile benchmark suite offers an easy-to-use way of quantifying QOE. Hence, this initial step was necessary to provide a method of identifying energy saving potential unused by existing DVFS techniques and to evaluate future DVFS improvements. The second study presented how an *Oracle* frequency profile for the benchmark workload was generated which serves as an energy and QOE baseline. Experiments comparing current mobile frequency governors to this baseline successfully revealed unused energy saving potential. The last two studies focused on optimising DVFS energy consumption. First, a runtime heuristic is developed to capture information on user perceived workload periods. A reinforcement learning based frequency governor then used this information to find improved frequency settings compared to standard approaches. The following sections will present more details on each study.

8.2.1 QOE Benchmarking Method

Initially, Chapter 4 described the development of a method to benchmark QOE for an interactive mobile workload. This method considers the user's point of view by analysing videos of screen output during workload executions. In those videos start and end frames of interactions are marked. The start frame indicates when the user provides input and the end frame marks when the user feels that the system has now finished processing the input. Those two frames allow measuring the duration of so called interaction lags for a workload execution with a given system configuration. With this method markups of different executions of the same workload can be compared to understand how different system configurations affect user perceived performance. By capturing interactions from real users, 16 mobile usage scenarios with a total length of 190 minutes were recorded. Together with the lag marking method, these recordings comprise an interactive benchmark for mobile systems to evaluate QOE. It can be used by researchers and industry alike to optimise their systems. Additionally, the presented method allows others to generate additional workload scenarios to extend or specialise the benchmark.

8.2.2 *Oracle* Limit Study

Chapter 5 presented how the QOE benchmark was used to create a baseline frequency profile which achieves maximum energy savings without perceptible performance impact. Initially, the chapter introduced a user irritation metric which gives an overall score to a workload execution by considering interaction lag markups. It sets execution deadlines for each interaction and quantifies deadline violations with an irritation penalty. The accumulated penalty of all lags in a workload indicates overall user irritation with the workload execution. In a second step, an *Oracle* algorithm used a comprehensive set of execution statistics for each available CPU frequency level to create the most energy efficiency and least irritating frequency profile for the benchmark. Lastly, this chapter described how the *Oracle* frequency profile was used to evaluate current standard frequency governors on mobile systems. It analysed their performance in terms of energy efficiency and user irritation. Frequency choices made by the *Ondemand*, *Interactive* and *Conservative* governor were compared to the *Oracle* baseline. Experimental results showed that *Interactive* and *Ondemand* achieve low user irritation but need on average 32% and 27% more energy than the *Oracle*. *Conservative* needs on average only 4% more energy but is irritating for as much as 32% of the time while the user is waiting for the system to respond to his input. This study could successfully provide evidence to support the motivational observations of how governors waste energy. It showed that there is potential to improve DVFS energy efficiency when considering the user's perspective.

8.2.3 Capturing the User's Point of View at Runtime

Until now the user's perspective was considered by evaluating video recordings of workload executions offline. To implement an improved DVFS governor which makes use of information on the user's perspective, a feasible online method was needed. A third study in Chapter 6 analysed system statistics of interaction lag executions to find correlations to lag dimensions as determined by video markups. Screen refresh executions of the *Android* display subsystem component *SurfaceFlinger* were determined to be a good runtime indicator for the user's point of view. Together with input data and CPU load, a heuristic was developed which can capture interaction lag dimensions as seen by the user. This heuristic is able to catch the last screen refresh in a lag, i.e. the user perceived lag ending, with an average error of 11.7%. As results of the final

study showed, the heuristic is accurate enough to improve energy efficiency compared to standard governors whilst providing good QOE to the end user.

8.2.4 Perception Aware DVFS Governor

The final study of this thesis in Chapter 7 presented an improved DVFS governor. It makes use of the lag end detection heuristic to identify workload periods as perceived by the end user. Based on this information the governor decides when to switch frequency settings. It optimises the balance between energy consumption and user irritation for interaction lags by using a reinforcement learning approach. In so doing, it is able to learn good behaviour for previously unseen interactions and to adapt to potential changes of known ones. The QOE benchmark and the *Oracle* baseline were used to evaluate energy efficiency and irritation of the improved DVFS technique. After an initial warm-up, the user perception aware governor is able to find good frequency settings for 93% of all evaluated interactions. It was able to beat *Interactive* and *Ondemand* in terms of energy consumption by needing up to 22% less energy, which is 9.6% more than the all knowing *Oracle* consumes. Also, it achieves 26.5% lower user irritation than *Conservative* which comes as close as a 5.6% difference to the *Oracle*.

8.3 Critical Analysis

The reinforcement learning governor developed in the last chapter is able to show improved energy efficiency compared to current standard approaches while maintaining a similar QOE. It is, however, unable to exactly achieve the *Oracle*'s results. The following reasons for this shortcoming were identified and will be addressed among other things in future work:

The benchmark workload generated in Chapter 4 covers enough interaction examples to reveal energy saving potential by evaluating current standard frequency governors against a baseline *Oracle*. Training and fine tuning the reinforcement learning governor, however, would benefit from a longer workload with more interactions. For the evaluation experiments in Chapter 7 the workload was executed multiple times. This way the governor could sample each interaction class often enough to learn good frequency settings. With a longer workload, multiple executions would not be neces-

sary and results would be more representative. In the future, the methodology from Chapter 4 can be used to collect this additional data.

Wrong classifications of interactions into interaction classes were preventing the governor from reaching *Oracle* results. If instances showing different behaviour and requiring different frequencies for low energy efficiency and user irritation, are assigned to the same class, the reinforcement learning governor is unable to optimise its frequency choice. For the experiments in Chapter 7 this classification was done manually where mistakes might have happened. An actual implementation of the interaction classification as described in Section 7.3 can help to avoid those mistakes.

The warm-up time to find good frequency settings for an interaction class can potentially be reduced. Statistical results indicated that only a few samples for different frequencies might be enough to calculate a scaling curve which is accurate enough to predict the optimal frequency. In Section 7.6.3 scaling sample statistics were used to estimate results for other frequencies. Estimation was, however, not yet accurate enough for a reliable prediction. Again, a longer workload containing more samples can help to achieve this and thereby reduce training time.

The results of Chapter 5 indicate that simply hard-wiring the CPU to a fixed frequency of 1.57 or 1.50 GHz would achive energy and irritation results better than any tested governor. This includes the reinforcement learning governor developed in this work even if only by a slight margin. Given the large body of research work on DVFS it is, however, highly unlikely that this simple solution is sufficient for modern mobile systems. The good performance of some fixed frequencies is more likely highlighting the limitations of the generated benchmark workload. Again an extended workload with longer usage recordings and a larger number of interaction classes would help to capture a more realistic picture.

Next to the improvements mentioned above other future work is planned based on the results of this study. It will be presented in the next section.

8.4 Future Work

Future work focuses on three areas: firstly, on extending the workload generation methodology to reduce limitations and capture more workload scenarios, secondly on

improving *RLGov* and thirdly on applying developed tools and methods to heterogeneous processing problems. This section will briefly describe each area.

Workload Generation The workload generation method introduced in Chapter 7 is unable to capture networking based workloads since they introduce a high level of non determinism. Executing an interaction on a website in the morning might lead to very different results compared to executing it in the evening. The resulting video frame showing the lag end can therefore be impossible to anticipate and the lag marking method fails. A recently released interaction record and replay tool [130] seems to handle networking workloads well. It has, however, no capabilities of benchmarking QOE by considering interaction lags. For the future, an extension of the lag marker method is planned considering mechanics of the mentioned tool. The goal is to include networking workloads and workloads using sensor readings such as camera, microphone or GPS. Additionally, it is planned to improve capturing the user perspective by including workloads that are dominated by *Jank* [180] type lags. During Jank, frames are dropped when the processor is too busy to keep up with the load. These occur mainly during CPU intensive workloads such as games, video playback or complex web page rendering.

RLGov *RLGov*'s current implementation and experiments executed on a workload simulator serve as a proof of concept that applied techniques can be used to reduce energy consumption. An actual implementation on mobile hardware is planned for the future. It will be evaluated against the benchmark workloads generated with the lag marker methodology. Additionally, a study is planned to execute the governor over long periods of time during day to day activities of actual users. Collected statistics of this study will contain valuable information on how the governor performs in the field and how it can be improved further.

Heterogeneous Processing Over the last years many research studies were conducted on heterogeneous processing technologies as a means of saving energy on mobile systems. A heterogeneous processor consists of two or more cores with different micro-architectural features such as different pipeline lengths, different cache sizes, different frequency scales, etc. A well known example is *Arm*'s *big.LITTLE* processor architecture [182]. The idea behind this technology is to provide two different core types with different performance capabilities and different energy profiles. Large and powerful cores are intended to be used for

performance intensive tasks and small energy efficient cores for low performance ones. It leaves the software developer with the task of implementing an intelligent scheduler which is able to find a good thread to core mapping. Methodologies and tools implemented in this study to improve DVFS can help in solving such heterogeneous scheduling problems. The QOE benchmark can evaluate scheduling techniques and demonstrate their effect on QOE. Frequency selection algorithms introduced for *RLGov* can be extended to not only find a good frequency level but also an optimal lag to core mapping. For future research it is planned to tackle heterogeneous scheduling problems by making use of the concepts developed in this thesis.

8.5 Final Remarks

This thesis could successfully show that information on the user's perspective can be used to improve energy efficiency of current DVFS techniques for interactive mobile workloads whilst delivering sufficient performance to satisfy the end user. This was demonstrated with the user perception aware frequency governor developed in Chapter 7. The results presented in that chapter do not fully exploit the energy saving potential identified by the *Oracle* study. They do, however, serve as a proof of concept. Developed methods lay a foundation for further improvements of perception aware DVFS. Additionally, they can be applied to evaluate and optimise techniques in related areas such as heterogeneous processing.

Bibliography

[1] *Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016*. URL: http://www.gartner.com/newsroom/id/3415117 (visited on 09/12/2016).

[2] Aaron Carroll and Gernot Heiser. "An Analysis of Power Consumption in a Smartphone". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 21–21. URL: http://dl.acm.org/citation.cfm?id=1855840.1855861 (visited on 12/09/2016).

[3] Xiang Chen et al. "How is Energy Consumed in Smartphone Display Applications?" In: *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. HotMobile '13. Jekyll Island, Georgia: ACM, 2013, 3:1–3:6. ISBN: 978-1-4503-1421-3. DOI: 10.1145/2444776.2444781.

[4] Marco Torchiano et al. "Profiling Power Consumption on Mobile Devices". In: *Proceedings of The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. ENERGY'13. Mar. 24, 2013, pp. 101–106. URL: https://www.researchgate.net/publication/236593462_Profiling_Power_Consumption_on_Mobile_Devices (visited on 10/21/2016).

[5] Sergey Zhuravlev et al. "Survey of Energy-Cognizant Scheduling Techniques". In: *IEEE Transactions on Parallel and Distributed Systems* 24.7 (July 2013), pp. 1447–1464. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.20.

[6] Olaf Kohlisch and Werner Kuhmann. "System response time and readiness for task execution the optimum duration of inter-task delays". In: *Ergonomics* 40.3 (Mar. 1, 1997), pp. 265–280. ISSN: 0014-0139. DOI: 10.1080/001401397188143.

[7] Alex Shye et al. “Learning and Leveraging the Relationship Between Architecture-Level Measurements and Individual User Satisfaction”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 427–438. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.29.

[8] S. Bischoff, A. Hansson, and B. M. Al-Hashimi. “Applying of Quality of Experience to system optimisation”. In: *Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation*. PATMOS’13. Sept. 2013, pp. 91–98. DOI: 10.1109/PATMOS.2013.6662160.

[9] Le Yan, Lin Zhong, and Niraj K. Jha. “User-perceived Latency Driven Voltage Scaling for Interactive Applications”. In: *Proceedings of the 42nd Annual Design Automation Conference*. DAC ’05. Anaheim, California, USA: ACM, 2005, pp. 624–627. ISBN: 1-59593-058-2. DOI: 10.1145/1065579.1065743.

[10] Jim Dabrowski and Ethan V. Munson. “40 years of searching for the best computer system response time”. In: *Interacting with Computers* 23.5 (Sept. 1, 2011), pp. 555–564. ISSN: 0953-5438, 1873-7951. DOI: 10.1016/j.intcom.2011.05.008.

[11] James R. Dabrowski and Ethan V. Munson. “Is 100 Milliseconds Too Fast?” In: *Proceedings of Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’01. Seattle, Washington: ACM, 2001, pp. 317–318. ISBN: 1-58113-340-5. DOI: 10.1145/634067.634255.

[12] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 978-0-201-16505-0.

[13] Steven C. Seow. *Designing and Engineering Time: The Psychology of Time Perception in Software*. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2008. 224 pp. ISBN: 978-0-321-50918-5.

[14] Trevor Mudge. “Power: A First-Class Architectural Design Constraint”. In: *Computer* 34.4 (Apr. 2001), pp. 52–58. ISSN: 0018-9162. DOI: 10.1109/2.917539.

[15] Vasanth Venkatachalam and Michael Franz. “Power Reduction Techniques for Microprocessor Systems”. In: *ACM Computing Surveys* 37.3 (Sept. 2005), pp. 195–237. ISSN: 0360-0300. DOI: 10.1145/1108956.1108957.

[16] Sparsh Mittal. “A survey of techniques for improving energy efficiency in embedded computing systems”. In: *International Journal of Computer Aided Engineering and Technology* 6.4 (Jan. 1, 2014), pp. 440–459. ISSN: 1757-2657. DOI: 10.1504/IJCAET.2014.065419.

[17] Mark Weiser et al. “Scheduling for Reduced CPU Energy”. In: *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*. OSDI ’94. Monterey, California: USENIX Association, 1994. URL: http://dl.acm.org/citation.cfm?id=1267638.1267640 (visited on 12/09/2016).

[18] Kinshuk Govil, Edwin Chan, and Hal Wasserman. “Comparing Algorithm for Dynamic Speed-setting of a Low-power CPU”. In: *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*. MobiCom ’95. New York, NY, USA: ACM, 1995, pp. 13–25. ISBN: 0-89791-814-2. DOI: 10.1145/215530.215546.

[19] Trevor Pering, Tom Burd, and Robert Brodersen. “The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms”. In: *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*. ISLPED ’98. New York, NY, USA: ACM, 1998, pp. 76–81. ISBN: 978-1-58113-059-1. DOI: 10.1145/280756.280790.

[20] Canturk Isci et al. “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–358. ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.8.

[21] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. “Automatic Performance Setting for Dynamic Voltage Scaling”. In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. MobiCom ’01. New York, NY, USA: ACM, 2001, pp. 260–271. ISBN: 1-58113-422-3. DOI: 10.1145/381677.381702.

[22] Andreas Weissel and Frank Bellosa. “Process Cruise Control: Event-driven Clock Scaling for Dynamic Power Management”. In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES ’02. New York, NY, USA: ACM, 2002, pp. 238–246. ISBN: 978-1-58113-575-6. DOI: 10.1145/581630.581668.

[23] Seongsoo Lee and Takayasu Sakurai. “Run-time Voltage Hopping for Low-power Real-time Systems”. In: *Proceedings of the 37th Annual Design Automation Conference*. DAC ’00. Los Angeles, California, USA: ACM, 2000, pp. 806–809. ISBN: 1-58113-187-9. DOI: 10.1145/337292.337785.

[24] Flavius Gruian. “Hard Real-time Scheduling for Low-energy Using Stochastic Data and DVS Processors”. In: *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. ISLPED ’01. New York, NY, USA: ACM, 2001, pp. 46–51. ISBN: 978-1-58113-371-4. DOI: 10.1145/383082.383092.

[25] Jacob R. Lorch and Alan Jay Smith. “Improving Dynamic Voltage Scaling Algorithms with PACE”. In: *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’01. New York, NY, USA: ACM, 2001, pp. 50–61. ISBN: 978-1-58113-334-9. DOI: 10.1145/378420.378429.

[26] Ajay Dudani, Frank Mueller, and Yifan Zhu. “Energy-conserving Feedback EDF Scheduling for Embedded Systems with Real-time Constraints”. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. LCTES/SCOPES ’02. New York, NY, USA: ACM, 2002, pp. 213–222. ISBN: 978-1-58113-527-5. DOI: 10.1145/513829.513865.

[27] Wanghong Yuan and Klara Nahrstedt. “Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 149–163. ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945460.

[28] Dongkun Shin, Jihong Kim, and Seongsoo Lee. “Low-energy Intra-task Voltage Scheduling Using Static Timing Analysis”. In: *Proceedings of the 38th Annual Design Automation Conference*. DAC ’01. New York, NY, USA: ACM, 2001, pp. 438–443. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.378551.

[29] A. Azevedo et al. “Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 168–. ISBN: 978-0-7695-1471-0. URL: http://dl.acm.org/citation.cfm?id=882452.874373 (visited on 09/21/2016).

[30] Chung-Hsing Hsu and Ulrich Kremer. “The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI ’03. New York, NY, USA: ACM, 2003, pp. 38–48. ISBN: 978-1-58113-662-3. DOI: 10.1145/781131.781137.

[31] F. Yao, A. Demers, and S. Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. FOCS ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–. ISBN: 0-8186-7183-1. URL: http://dl.acm.org/citation.cfm?id=795662.796264 (visited on 12/09/2016).

[32] Tohru Ishihara and Hiroto Yasuura. “Voltage Scheduling Problem for Dynamically Variable Voltage Processors”. In: *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*. ISLPED ’98. New York, NY, USA: ACM, 1998, pp. 197–202. ISBN: 978-1-58113-059-1. DOI: 10.1145/280756.280894.

[33] Hakan Aydi et al. “Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems”. In: *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*. RTSS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 95–. ISBN: 0-7695-1420-0. URL: http://dl.acm.org/citation.cfm?id=882482.883797 (visited on 12/09/2016).

[34] Gang Quan and Xiaobo Hu. “Energy Efficient Fixed-priority Scheduling for Real-time Systems on Variable Voltage Processors”. In: *Proceedings of the 38th Annual Design Automation Conference*. DAC ’01. Las Vegas, Nevada, USA: ACM, 2001, pp. 828–833. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379074.

[35] Saowanee Saewong and Ragunathan (Raj) Rajkumar. “Practical Voltage-Scaling for Fixed-Priority RT-Systems”. In: *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 106–. ISBN: 0-7695-1956-3. DOI: 10.1109/RTTAS.2003.1203042.

[36] D. Zhu, R. Melhem, and B. R. Childers. “Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 14.7 (July 2003), pp. 686–700. ISSN: 1045-9219. DOI: 10.1109/TPDS.2003.1214320.

[37] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. "Energy-aware Scheduling for Real-time Multiprocessor Systems with Uncertain Task Execution Time". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. New York, NY, USA: ACM, 2007, pp. 664–669. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278648.

[38] Phillip Stanley-Marbell, Michael S. Hsiao, and Ulrich Kremer. "A Hardware Architecture for Dynamic Performance and Energy Adaptation". In: *Lecture Notes in Computer Science*. 2325. Springer Berlin Heidelberg, Feb. 2, 2002, pp. 33–52. ISBN: 978-3-540-01028-9 978-3-540-36612-6. DOI: 10.1007/3-540-36612-1_3.

[39] Kihwan Choi, R. Soma, and M. Pedram. "Fine-grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Tradeoff Based on the Ratio of Off-chip Access to On-chip Computation Times". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.1 (Nov. 2006), pp. 18–28. ISSN: 0278-0070. DOI: 10.1109/TCAD.2004.839485(410)24.

[40] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. "Dynamic Voltage and Frequency Scaling Based on Workload Decomposition". In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ISLPED '04. New York, NY, USA: ACM, 2004, pp. 174–179. ISBN: 1-58113-929-2. DOI: 10.1145/1013235.1013282.

[41] Masaaki Kondo and Hiroshi Nakamura. "Dynamic Processor Throttling for Power Efficient Computations". In: *Proceedings of the 4th International Conference on Power-Aware Computer Systems*. PACS'04. Portland, OR: Springer-Verlag, 2005, pp. 120–134. ISBN: 3-540-29790-1, 978-3-540-29790-1. DOI: 10.1007/11574859_9.

[42] Chung-Hsing Hsu and Wu-Chun Feng. "Effective Dynamic Voltage Scaling Through CPU-Boundedness Detection". In: *Proceedings of the 4th International Conference on Power-Aware Computer Systems*. PACS'04. Portland, OR: Springer-Verlag, 2005, pp. 135–149. ISBN: 3-540-29790-1, 978-3-540-29790-1. DOI: 10.1007/11574859_10.

[43] V. W. Freeh et al. "Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications". In: *IEEE Transactions on Parallel and Distributed*

Systems 18.6 (June 2007), pp. 835–848. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.1026.

[44] Daniel Shelepov et al. "HASS: A Scheduler for Heterogeneous Multicore Systems". In: *ACM SIGOPS Operating Systems Review* 43.2 (Apr. 2009), pp. 66–75. ISSN: 0163-5980. DOI: 10.1145/1531793.1531804.

[45] G. Dhiman and T. Rosing. "System-Level Power Management Using Online Learning". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.5 (May 2009), pp. 676–689. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2015740.

[46] Juan Carlos Saez et al. "A Comprehensive Scheduler for Asymmetric Multicore Systems". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 139–152. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755929.

[47] Greg Semeraro et al. "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling". In: *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 29–. DOI: 10.1109/HPCA.2002.995696.

[48] Anoop Iyer and Diana Marculescu. "Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores". In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '02. San Jose, California: ACM, 2002, pp. 379–386. ISBN: 0-7803-7607-2. DOI: 10.1145/774572.774629.

[49] Anoop Iyer and Diana Marculescu. "Microarchitecture-level Power Management". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.3 (June 2002), pp. 230–239. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2002.1043326.

[50] Greg Semeraro et al. "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture". In: *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 35. Istanbul, Turkey: IEEE Computer Society Press, 2002, pp. 356–367. ISBN: 0-7695-1859-1. URL: http://dl.acm.org/citation.cfm?id=774861.774899 (visited on 12/09/2016).

[51] Grigorios Magklis et al. “Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor”. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA ’03. San Diego, California: ACM, 2003, pp. 14–27. ISBN: 0-7695-1945-8. DOI: 10.1145/859618.859621. URL: http://doi.acm.org/10.1145/859618.859621.

[52] Steven Dropsho et al. “Dynamically Trading Frequency for Complexity in a GALS Microprocessor”. In: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 157–168. ISBN: 978-0-7695-2126-8. DOI: 10.1109/MICRO.2004.18.

[53] Diana Marculescu. “Application Adaptive Energy Efficient Clustered Architectures”. In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ISLPED ’04. New York, NY, USA: ACM, 2004, pp. 344–349. ISBN: 978-1-58113-929-7. DOI: 10.1145/1013235.1013318.

[54] Emil Talpes and Diana Marculescu. “Toward a Multiple Clock/Voltage Island Design Style for Power-aware Processors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13.5 (May 2005), pp. 591–603. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2005.844305.

[55] Sebastian Herbert and Diana Marculescu. “Analysis of Dynamic Voltage/Frequency Scaling in Chip-multiprocessors”. In: *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*. ISLPED ’07. New York, NY, USA: ACM, 2007, pp. 38–43. ISBN: 978-1-59593-709-4. DOI: 10.1145/1283780.1283790.

[56] S. Herbert and D. Marculescu. “Variation-aware dynamic voltage/frequency scaling”. In: *Proceedings of the 15th International Symposium on High Performance Computer Architecture*. HPCA ’09. Feb. 2009, pp. 301–312. DOI: 10.1109/HPCA.2009.4798265.

[57] Dongkun Shin and Jihong Kim. “Power-aware Scheduling of Conditional Task Graphs in Real-time Multiprocessor Systems”. In: *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. ISLPED ’03. New York, NY, USA: ACM, 2003, pp. 408–413. ISBN: 978-1-58113-682-1. DOI: 10.1145/871506.871607.

[58] I. Kadayif, M. Kandemir, and I. Kolcu. "Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption in Chip Multiprocessors". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21158–. ISBN: 0-7695-2085-5. DOI: 10.1109/DATE.2004.1269048.

[59] Matthew Curtis-Maury et al. "Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. New York, NY, USA: ACM, 2006, pp. 157–166. ISBN: 978-1-59593-282-2. DOI: 10.1145/1183401.1183426.

[60] Wonyoung Kim et al. "System level analysis of fast, per-core DVFS using on-chip switching regulators". In: *Proceedings of the 14th International Symposium on High Performance Computer Architecture*. HPCA '08. Feb. 2008, pp. 123–134. DOI: 10.1109/HPCA.2008.4658633.

[61] Radu Teodorescu and Josep Torrellas. "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 363–374. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.40.

[62] Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. "Improving Fairness, Throughput and Energy-efficiency on a Chip Multiprocessor Through DVFS". In: *ACM SIGARCH Computer Architecture News* 35.1 (Mar. 2007), pp. 31–38. ISSN: 0163-5964. DOI: 10.1145/1241601.1241609.

[63] R. Watanabe et al. "Power reduction of chip multi-processors using shared resource control cooperating with DVFS". In: *Proceedings of the 25th International Conference on Computer Design*. ICCD '07. Oct. 2007, pp. 615–622. DOI: 10.1109/ICCD.2007.4601961.

[64] Noriko Takagi et al. "Cooperative Shared Resource Access Control for Low-power Chip Multiprocessors". In: *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '09. New York, NY, USA: ACM, 2009, pp. 177–182. ISBN: 978-1-60558-684-7. DOI: 10.1145/1594233.1594278.

[65] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. “vGreen: A System for Energy Efficient Computing in Virtualized Environments”. In: *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’09. New York, NY, USA: ACM, 2009, pp. 243–248. ISBN: 978-1-60558-684-7. DOI: 10.1145/1594233.1594292.

[66] S. Cho and R. Melhem. “Corollaries to Amdahl’s Law for Energy”. In: *IEEE Computer Architecture Letters* 7.1 (Jan. 2008), pp. 25–28. ISSN: 1556-6056. DOI: 10.1109/L-CA.2007.18.

[67] S. Cho and R. G. Melhem. “On the Interplay of Parallelization, Program Performance, and Energy Consumption”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.3 (Mar. 2010), pp. 342–353. ISSN: 1045-9219. DOI: 10.1109/TPDS.2009.41.

[68] Etienne Le Sueur and Gernot Heiser. “Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns”. In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. HotPower ’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. URL: http://dl.acm.org/citation.cfm?id=1924920.1924921 (visited on 09/16/2016).

[69] Youngbin Seo, Jeongki Kim, and Euiseong Seo. “Effectiveness Analysis of DVFS and DPM in Mobile Devices”. In: *Journal of Computer Science and Technology* 27.4 (July 1, 2012), pp. 781–790. ISSN: 1000-9000, 1860-4749. DOI: 10.1007/s11390-012-1264-6.

[70] Robert B. Miller. “Response Time in Man-computer Conversational Transactions”. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS ’68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. DOI: 10.1145/1476589.1476628.

[71] M. Grossberg, R. A. Wiesen, and D. B. Yntema. “An Experiment on Problem Solving with Delayed Computer Responses”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-6.3 (Mar. 1976), pp. 219–222. ISSN: 0018-9472. DOI: 10.1109/TSMC.1976.5409241.

[72] Robert C. Williges and Beverly H. Williges. “Modeling the Human Operator in Computer-Based Data Entry”. In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 24.3 (June 1, 1982), pp. 285–299. ISSN: 0018-7208, 1547-8181. DOI: 10.1177/001872088202400304.

[73] Raymond E. Barber and Henry C. Lucas Jr. “System Response Time Operator Productivity, and Job Satisfaction”. In: *Communications of the ACM* 26.11 (Nov. 1983), pp. 972–986. ISSN: 0001-0782. DOI: 10.1145/182.358464.

[74] T. W. Butler. “Computer Response Time and User Performance.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’83. New York, NY, USA: ACM, 1983, pp. 58–62. ISBN: 978-0-89791-121-4. DOI: 10.1145/800045.801581.

[75] Gary L Dannenbring. “The effect of computer response time on user performance and satisfaction: A preliminary investigation”. In: *Behavior Research Methods & Instrumentation* 15.2 (1983), pp. 213–216.

[76] Gale L. Martin and Kenneth G. Corl. “System response time effects on user productivity”. In: *Behaviour & Information Technology* 5.1 (Jan. 1, 1986), pp. 3–13. ISSN: 0144-929X. DOI: 10.1080/01449298608914494.

[77] Florian Schaefer. “The Effect of System Response Times on Temporal Predictability of Work Flow in Human-Computer Interaction”. In: *Human Performance* 3.3 (Sept. 1, 1990), pp. 173–186. ISSN: 0895-9285. DOI: 10.1207/s15327043hup0303_3.

[78] Steven L. Teal and Alexander I. Rudnicky. “A Performance Model of System Delay and User Strategy Selection”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’92. New York, NY, USA: ACM, 1992, pp. 295–305. ISBN: 978-0-89791-513-7. DOI: 10.1145/142750.142818.

[79] I. Scott MacKenzie and Colin Ware. “Lag As a Determinant of Human Performance in Interactive Systems”. In: *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*. CHI ’93. New York, NY, USA: ACM, 1993, pp. 488–493. ISBN: 978-0-89791-575-5. DOI: 10.1145/169059.169431.

[80] M. Thum et al. “Standardized task strain and system response times in human-computer interaction”. In: *Ergonomics* 38.7 (July 1995), pp. 1342–1351. ISSN: 0014-0139. DOI: 10.1080/00140139508925192.

[81] Paddy O’Donnell and Stephen W. Draper. “How Machine Delays Change User Strategies”. In: *ACM SIGCHI Bulletin* 28.2 (Apr. 1996), pp. 39–42. ISSN: 0736-6906. DOI: 10.1145/226650.226665.

[82] T. Goodman and R. Spence. “The Effect of System Response Time on Interactive Computer Aided Problem Solving”. In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’78. New York, NY, USA: ACM, 1978, pp. 100–104. DOI: 10.1145/800248.807378.

[83] T. J. Goodman and R. Spence. “The Effect of Computer System Response Time Variability on Interactive Graphical Problem Solving”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.3 (Mar. 1981), pp. 207–216. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308654.

[84] Tom Goodman and Robert Spence. “The Effects of Potentiometer Dimensionality, System Response Time, and Time of Day on Interactive Graphical Problem Solving”. In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 24.4 (Aug. 1, 1982), pp. 437–456. ISSN: 0018-7208, 1547-8181. DOI: 10.1177/001872088202400406.

[85] Lawrence J Kosmatka. “A user challenges value of subsecond response time”. In: *Computerworld* 18.24 (1984), pp. 1–8.

[86] G. N. Lambert. “A Comparative Study of System Response Time on Program Developer Productivity”. In: *IBM Systems Journal* 23.1 (Mar. 1984), pp. 36–43. ISSN: 0018-8670. DOI: 10.1147/sj.231.0036.

[87] John D Gould, Clayton Lewis, and Vincent Barnes. “Cursor movement during text editing”. In: *ACM Transactions on Information Systems (TOIS)* 3.1 (1985), pp. 22–34.

[88] William C. Treurniet, Paul J. Hearty, and Miguel A. Planas. “Viewers’ responses to delays in simulated teletext reception”. In: *Behaviour & Information Technology* 4.3 (July 1, 1985), pp. 177–188. ISSN: 0144-929X. DOI: 10.1080/01449298508901799.

[89] Jan L. Guynes. “Impact of System Response Time on State Anxiety”. In: *Commun. ACM* 31.3 (Mar. 1988), pp. 342–347. ISSN: 0001-0782. DOI: 10.1145/42392.42402.

[90] Paula M. Van Balen and Leslie R. Eisler. “Evaluation of Audio Response Time Delay Requirements for Digitized Audio”. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 33.4 (Oct. 1, 1989), pp. 234–238. ISSN: 1541-9312, DOI: 10.1177/154193128903300407.

[91] Thomas W Butler. “Computer response time and user performance during data entry”. In: *AT&T Bell Laboratories Technical Journal* 63.6 (1984), pp. 1007–1018.

[92] Alan Dix. “Pace and interaction”. In: *People and computers* (1992), pp. 193–193.

[93] Kenton O’Hara. “Cost of Operations Affects Planfulness of Problem-solving Behaviour”. In: *Conference Companion on Human Factors in Computing Systems*. CHI ’94. New York, NY, USA: ACM, 1994, pp. 105–106. ISBN: 978-0-89791-651-6. DOI: 10.1145/259963.260083.

[94] Miguel A Planas and William C Treurniet. “The effects of feedback during delays in simulated teletext reception”. In: *Behaviour & Information Technology* 7.2 (1988), pp. 183–191.

[95] Lawrence M. Schleifer and Benjamin C. Amick III. “System response time and method of pay: Stress effects in computer-based tasks”. In: *International Journal of Human-Computer Interaction* 1.1 (Jan. 1, 1989), pp. 23–39. ISSN: 1044-7318. DOI: 10.1080/10447318909525955.

[96] Joachim Meyer et al. “Duration estimates and users’ preferences in human-computer interaction”. In: *Ergonomics* 39.1 (1996), pp. 46–60.

[97] Andrew Sears, Julie A. Jacko, and Michael S. Borella. “Internet Delay Effects: How Users Perceive Quality, Organization, and Ease of Use of Information”. In: *CHI ’97 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’97. Atlanta, Georgia: ACM, 1997, pp. 353–354. ISBN: 0-89791-926-2. DOI: 10.1145/1120212.1120430.

[98] Judith Ramsay, Alessandro Barbesi, and Jenny Preece. “A psychological investigation of long retrieval times on the World Wide Web”. In: *Interacting with computers* 10.1 (1998), pp. 77–86.

[99] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. “Integrating User-perceived Quality into Web Server Design”. In: *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 2000, pp. 1–16. URL: http://dl.acm.org/citation.cfm?id=347319.346245 (visited on 09/25/2016).

[100] John A Hoxmeier and Chris DiCesare. “System response time and user satisfaction: An experimental study of browser-based applications”. In: *AMCIS 2000 Proceedings* (2000), p. 347.

[101] Paula R Selvidge, Barbara S Chaparro, and Gregory T Bender. “The world wide wait: effects of delays on user performance”. In: *International Journal of Industrial Ergonomics* 29.1 (2002), pp. 15–20.

[102] Fiona Fui-Hoon Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163.

[103] Arnout RH Fischer, Frans JJ Blommaert, and Cees JH Midden. “Monitoring and evaluation of time delay”. In: *International Journal of Human-Computer Interaction* 19.2 (2005), pp. 163–180.

[104] Werner Kuhmann et al. “Experimental investigation of psychophysiological stress-reactions induced by different system response times in human-computer interaction”. In: *Ergonomics* 30.6 (1987), pp. 933–943.

[105] Werner Kuhmann. “Experimental investigation of stress-inducing properties of system response times”. In: *Ergonomics* 32.3 (1989), pp. 271–280.

[106] Olaf Kohlisch and Florian Schaefer. “Physiological changes during computer tasks: responses to mental load or to motor demands?” In: *Ergonomics* 39.2 (1996), pp. 213–224.

[107] Glen Anderson, Rina Doherty, and Eric Baugh. “Diminishing Returns?: Revisiting Perception of Computing Performance”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’11. New York, NY, USA: ACM, 2011, pp. 2703–2706. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979338.

[108] Rina A. Doherty and Paul Sorenson. “Keeping Users in the Flow: Mapping System Responsiveness with User Experience”. In: *Procedia Manufacturing*. 6th International Conference on Applied Human Factors and Ergonomics (AHFE 2015) and the Affiliated Conferences, AHFE 2015 3 (Jan. 1, 2015), pp. 4384–4391. ISSN: 2351-9789. DOI: 10.1016/j.promfg.2015.07.436.

[109] N. Tolia, D. G. Andersen, and M. Satyanarayanan. “Quantifying interactive user experience on thin clients”. In: *Computer* 39.3 (Mar. 2006), pp. 46–52. ISSN: 0018-9162. DOI: 10.1109/MC.2006.101.

[110] Ingmar Verheij. *Quantify Perceived Performance*. Sept. 20, 2011. URL: http://www.ingmarverheij.com/wp-content/uploads/downloads/2011/09/Whitepaper-Quantifying-Perceived-Performance-v1.0.pdf (visited on 09/13/2016).

[111] Timothy Mangan. *Perceived Performance : Tuning a system for what really matters*. Sept. 18, 2003. URL: http://www.tmurgent.com/WhitePapers/PerceivedPerformance.pdf (visited on 09/13/2016).

[112] Ernst Heinrich Weber. *EH Weber: The sense of touch*. (H. E. Ross & D. J. Murray, Trans., Original work published 1834). London: Academic Press for Experimental Psychology Society, 1978. 296 pp.

[113] Ben Shneiderman. "Response Time and Display Rate in Human Performance with Computers". In: *ACM Computing Surveys (CSUR)* 16.3 (Sept. 1984), pp. 265–285. ISSN: 0360-0300. DOI: 10.1145/2514.2517. URL: http://doi.acm.org/10.1145/2514.2517.

[114] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 978-0-262-19398-6.

[115] Martin Riedmiller et al. "Reinforcement learning for robot soccer". In: *Autonomous Robots* 27.1 (May 15, 2009), pp. 55–73. ISSN: 0929-5593, 1573-7527. DOI: 10.1007/s10514-009-9120-4.

[116] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68. URL: http://cling.csd.uwo.ca/cs346a/extra/tdgammon.pdf (visited on 10/07/2016).

[117] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 26, 2015), pp. 529–533. ISSN: 0028-0836. DOI: 10.1038/nature14236.

[118] *Android Open Source Project*. URL: https://source.android.com/ (visited on 11/21/2016).

[119] Venkatesh Pallipadi and Alexey Starikovskiy. "The ondemand governor". In: *Proceedings of the Linux Symposium*. Vol. 2. sn, 2006, pp. 215–230. URL: http://mathdesc.fr/documents/kernel/linuxsymposium_procv2.pdf#page=223 (visited on 09/16/2016).

[120] Mike Chan. *Linux Kernel Patch: cpufreq: interactive: New 'interactive' governor*. LWN.net. Oct. 28, 2015. URL: https://lwn.net/Articles/662209/ (visited on 11/21/2016).

[121] A. Memon et al. "The first decade of GUI ripping: Extensions, applications, and broader impacts". In: *Proceedings of the 20th Working Conference on Reverse Engineering*. WCRE '13. Oct. 2013, pp. 11–20. DOI: 10.1109/WCRE.2013.6671275.

[122] Henrik Sandklef. *GNU Xnee v. 3.15*. GNU Xnee. 2012. URL: https://xnee.wordpress.com/ (visited on 10/10/2016).

[123] Timothy Wall. *Abbot framework for automated testing of Java GUI components and programs*. Mar. 2014. URL: http://abbot.sourceforge.net/doc/overview.shtml (visited on 03/03/2014).

[124] *Google UI/Application Exerciser Monkey*. Mar. 2014. URL: http://developer.android.com/tools/help/monkey.html (visited on 03/03/2014).

[125] *Google MonkeyRunner*. Mar. 2014. URL: http://developer.android.com/tools/help/monkeyrunner_concepts.html (visited on 03/03/2014).

[126] *GUITAR - A GUI Testing Framework*. SourceForge. Mar. 2014. URL: http://sourceforge.net/projects/guitar/ (visited on 03/03/2014).

[127] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. "A GUI Crawling-Based Technique for Android Mobile Application Testing". In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 252–261. ISBN: 978-0-7695-4345-1. DOI: 10.1109/ICSTW.2011.77.

[128] Lorenzo Gomez et al. "RERAN: Timing- and Touch-sensitive Record and Replay for Android". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 72–81. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606553.

[129] M. Halpern et al. "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem". In: *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '15. Mar. 2015, pp. 215–224. DOI: 10.1109/ISPASS.2015.7095807.

[130] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. "Versatile Yet Lightweight Record-and-replay for Android". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 349–366. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814320.

[131] Anthony Gutierrez et al. "Full-system Analysis and Characterization of Interactive Smartphone Applications". In: *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 81–90. ISBN: 978-1-4577-2063-5. DOI: 10.1109/IISWC.2011.6114205.

[132] Yongbing Huang et al. "Moby: A Mobile Benchmark Suite for Architectural Simulators". In: *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '14. Mar. 2014. DOI: 10.1109/ISPASS.2014.6844460.

[133] D. Pandiyan, Shin-Ying Lee, and C.-J. Wu. "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - MobileBench". In: *Proceedings of the 2013 IEEE International Symposium on Workload Characterization*. IISWC '13. Sept. 2013, pp. 133–142. DOI: 10.1109/IISWC.2013.6704679.

[134] Sangwook Kim et al. "Empirical Analysis of Power Management Schemes for Multi-core Smartphones". In: *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '13. New York, NY, USA: ACM, 2013, 109:1–109:7. ISBN: 978-1-4503-1958-4. DOI: 10.1145/2448556.2448665.

[135] Dam Sunwoo et al. "A structured approach to the simulation, analysis and characterization of smartphone applications". In: *Proceedings of the 2013 IEEE International Symposium on Workload Characterization*. IISWC '13. Sept. 2013, pp. 113–122. DOI: 10.1109/IISWC.2013.6704677.

[136] Jon Froehlich et al. "MyExperience: A System for in Situ Tracing and Capturing of User Feedback on Mobile Phones". In: *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*. MobiSys '07. New York, NY, USA: ACM, 2007, pp. 57–70. ISBN: 978-1-59593-614-1. DOI: 10.1145/1247660.1247670.

[137] Hokwon Song et al. “Usage Pattern-based Prefetching: Quick Application Launch on Mobile Devices”. In: *Proceedings of the 12th International Conference on Computational Science and Its Applications - Volume Part III*. ICCSA ’12. Salvador de Bahia, Brazil: Springer-Verlag, 2012, pp. 227–237. ISBN: 978-3-642-31136-9. DOI: 10.1007/978-3-642-31137-6_17.

[138] Lide Zhang et al. “Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance”. In: *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’13. Piscataway, NJ, USA: IEEE Press, 2013, 33:1–33:10. ISBN: 978-1-4799-1417-3. DOI: 10.1109/CODES-ISSS.2013.6659020.

[139] Lenin Ravindranath et al. “AppInsight: Mobile App Performance Monitoring in the Wild”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 107–120. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387891 (visited on 01/12/2015).

[140] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. “Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, NY, USA: ACM, 2009, pp. 168–178. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669135.

[141] Hossein Falaki et al. “A First Look at Traffic on Smartphones”. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC ’10. New York, NY, USA: ACM, 2010, pp. 281–287. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879176.

[142] Hossein Falaki et al. “Diversity in Smartphone Usage”. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’10. New York, NY, USA: ACM, 2010, pp. 179–194. ISBN: 978-1-60558-985-5. DOI: 10.1145/1814433.1814453.

[143] Earl Oliver. “Diversity in Smartphone Energy Consumption”. In: *Proceedings of the 2010 ACM Workshop on Wireless of the Students, by the Students, for the Students*. S3 ’10. New York, NY, USA: ACM, 2010, pp. 25–28. ISBN: 978-1-4503-0144-2. DOI: 10.1145/1860039.1860048.

[144] Hossein Falaki, Ratul Mahajan, and Deborah Estrin. "SystemSens: A Tool for Monitoring Usage in Smartphone Research Deployments". In: *Proceedings of the Sixth International Workshop on MobiArch*. MobiArch '11. New York, NY, USA: ACM, 2011, pp. 25–30. ISBN: 978-1-4503-0740-6. DOI: 10.1145/1999916.1999923.

[145] Clayton Shepard et al. "LiveLab: Measuring Wireless Networks and Smartphone Users in the Field". In: *ACM SIGMETRICS Performance Evaluation Review* 38.3 (Jan. 2011), pp. 15–20. ISSN: 0163-5999. DOI: 10.1145/1925019.1925023.

[146] Niels Henze. "Hit It!: An Apparatus for Upscaling Mobile HCI Studies". In: *CHI '12 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '12. New York, NY, USA: ACM, 2012, pp. 1333–1338. ISBN: 978-1-4503-1016-1. DOI: 10.1145/2212776.2212450.

[147] Ahmad Rahmati et al. "Tales of 34 iPhone Users: How they change and why they are different". In: *arXiv:1106.5100 [cs]* (June 24, 2011). arXiv: 1106.5100. URL: http://arxiv.org/abs/1106.5100 (visited on 10/12/2016).

[148] *Flurry by Yahoo*. Flurry. URL: http://www.flurry.com/ (visited on 01/12/2015).

[149] *PreEmptive Analytics*. URL: http://www.preemptive.com/pa (visited on 01/12/2015).

[150] Trinh Minh Tri Do, Jan Blom, and Daniel Gatica-Perez. "Smartphone Usage in the Wild: A Large-scale Analysis of Applications and Context". In: *Proceedings of the 13th International Conference on Multimodal Interfaces*. ICMI '11. New York, NY, USA: ACM, 2011, pp. 353–360. ISBN: 978-1-4503-0641-6. DOI: 10.1145/2070481.2070550.

[151] Shahram Mohrehkesh et al. "Demographic Prediction of Mobile User from Phone Usage". In: *ResearchGate* (June 1, 2012). URL: https://www.researchgate.net/publication/264888907_Demographic_Prediction_of_Mobile_User_from_Phone_Usage (visited on 10/13/2016).

[152] Y. Zhu, M. Halpern, and V. J. Reddi. "Event-based scheduling for energy-efficient QoS (eQoS) in mobile Web applications". In: *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*. HPCA '15. Feb. 2015, pp. 137–149. DOI: 10.1109/HPCA.2015.7056028.

[153] Lenin Ravindranath et al. “Timecard: Controlling User-perceived Delays in Server-based Mobile Applications”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. New York, NY, USA: ACM, 2013, pp. 85–100. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522717.

[154] Xuetao Wei et al. “ProfileDroid: Multi-layer Profiling of Android Applications”. In: *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*. Mobicom ’12. New York, NY, USA: ACM, 2012, pp. 137–148. ISBN: 978-1-4503-1159-5. DOI: 10.1145/2348543.2348563.

[155] Feng Qian et al. “Profiling Resource Usage for Mobile Applications: A Cross-layer Approach”. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’11. New York, NY, USA: ACM, 2011, pp. 321–334. ISBN: 978-1-4503-0643-0. DOI: 10.1145/1999995.2000026.

[156] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. “Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof”. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys ’12. New York, NY, USA: ACM, 2012, pp. 29–42. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168841.

[157] Chanmin Yoon et al. “AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 36–36. URL: http://dl.acm.org/citation.cfm?id=2342821.2342857 (visited on 01/12/2015).

[158] Kitae Kim et al. “FEPMA: Fine-grained event-driven power meter for android smartphones based on device driver layer event monitoring”. In: *Proceedings of the 2014 Design, Automation and Test in Europe Conference and Exhibition*. DATE ’14. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE2014.380.

[159] Seokjun Lee, Chanmin Yoon, and Hojung Cha. “User Interaction-based Profiling System for Android Application Tuning”. In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. UbiComp ’14. New York, NY, USA: ACM, 2014, pp. 289–299. ISBN: 978-1-4503-2968-2. DOI: 10.1145/2632048.2636091.

[160] Ding Li et al. “Calculating Source Line Level Energy Information for Android Applications”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA ’13. New York, NY, USA: ACM, 2013, pp. 78–89. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2483780.

[161] Abhinav Pathak et al. “What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps”. In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’12. New York, NY, USA: ACM, 2012, pp. 267–280. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307661.

[162] Kwanghwan Kim and Hojung Cha. “WakeScope: Runtime WakeLock Anomaly Management Scheme for Android Platform”. In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. EMSOFT ’13. Piscataway, NJ, USA: IEEE Press, 2013, 27:1–27:10. ISBN: 978-1-4799-1443-2. URL: http://dl.acm.org/citation.cfm?id=2555754.2555781 (visited on 10/12/2016).

[163] J.R. Lorch and A.J. Smith. “Using user interface event information in dynamic voltage scaling algorithms”. In: *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*. MASCOTS ’03. Oct. 2003, pp. 46–55. DOI: 10.1109/MASCOT.2003.1240641.

[164] Lin Zhong and Niraj K. Jha. “Dynamic Power Optimization of Interactive Systems”. In: *Proceedings of the 17th International Conference on VLSI Design*. VLSID ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 1041–. ISBN: 0-7695-2072-3. DOI: 10.1109/ICVD.2004.1261067.

[165] L. Yan, Lin Zhong, and N.K. Jha. “User-perceived latency driven voltage scaling for interactive applications”. In: *Proceedings of the 42nd Annual Design Automation Conference*. DAC ’05. June 2005, pp. 624–627. DOI: 10.1109/DAC.2005.193886.

[166] A. Shye et al. “Power to the people: Leveraging human physiological traits to control microprocessor frequency”. In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. Nov. 2008, pp. 188–199. DOI: 10.1109/MICRO.2008.4771790.

[167] Arindam Mallik et al. “PICSEL: Measuring User-perceived Performance to Control Dynamic Frequency Scaling”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’08. New York, NY, USA: ACM, 2008, pp. 70–79. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346291.

[168] Wook Song et al. “Reducing Energy Consumption of Smartphones Using User-perceived Response Time Analysis”. In: *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. HotMobile ’14. New York, NY, USA: ACM, 2014, 20:1–20:6. ISBN: 978-1-4503-2742-8. DOI: 10.1145/2565585.2565595.

[169] Hwisung Jung and M. Pedram. “Supervised Learning Based Power Management for Multicore Processors”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.9 (Sept. 2010), pp. 1395–1408. ISSN: 0278-0070. DOI: 10.1109/TCAD.2010.2059270.

[170] Michael Moeng and Rami Melhem. “Applying Statistical Machine Learning to Multicore Voltage & Frequency Scaling”. In: *Proceedings of the 7th ACM International Conference on Computing Frontiers*. CF ’10. New York, NY, USA: ACM, 2010, pp. 277–286. ISBN: 978-1-4503-0044-5. DOI: 10.1145/1787275.1787336.

[171] B. Rountree et al. “Practical performance prediction under Dynamic Voltage Frequency Scaling”. In: *Proceedings of the 2nd International Green Computing Conference*. IGCC ’11. July 2011, pp. 1–8. DOI: 10.1109/IGCC.2011.6008553.

[172] Edward Y. Y. Kan, W. K. Chan, and T. H. Tse. “EClass: An execution classification approach to improving the energy-efficiency of software via machine learning”. In: *Journal of Systems and Software* 85.4 (Apr. 2012), pp. 960–973. ISSN: 0164-1212. DOI: 10.1016/j.jss.2011.11.1010.

[173] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. “Predicting Performance Impact of DVFS for Realistic Memory Systems”. In: *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 155–165. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.23.

[174] H. Shen, J. Lu, and Q. Qiu. “Learning based DVFS for simultaneous temperature, performance and energy management”. In: *Proceedings of the 13th International Symposium on Quality Electronic Design*. ISQED ’12. Mar. 2012, pp. 747–754. DOI: 10.1109/ISQED.2012.6187575.

[175] F. M. M. u Islam and M. Lin. “Hybrid DVFS Scheduling for Real-Time Systems Based on Reinforcement Learning”. In: *IEEE Systems Journal* PP.99 (2015), pp. 1–10. ISSN: 1932-8184. DOI: 10.1109/JSYST.2015.2446205.

[176] Ming-Feng Chang and Wen-Yew Liang. “Learning-Directed Dynamic Voltage and Frequency Scaling for Computation Time Prediction”. In: *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. TRUSTCOM ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1023–1029. ISBN: 978-0-7695-4600-1. DOI: 10.1109/TrustCom.2011.140.

[177] Xueliang Li et al. “SmartCap: User Experience-oriented Power Adaptation for Smartphone’s Application Processor”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’13. Grenoble, France: EDA Consortium, 2013, pp. 57–60. ISBN: 978-1-4503-2153-2. DOI: 10.7873/DATE.2013.026.

[178] *Game Capture HD*. elgato.com. URL: http://www.elgato.com/en/gaming/game-capture-hd (visited on 04/23/2014).

[179] *Growth of Time Spent on Mobile Devices Slows - eMarketer*. emarketer.com. Oct. 7, 2015. URL: http://www.emarketer.com/Article/Growth-of-Time-Spent-on-Mobile-Devices-Slows/1013072 (visited on 11/04/2016).

[180] *Google I/O 2013 - Jank Free: Chrome Rendering Performance*. In collab. with Google Developers. May 16, 2013. URL: http://www.youtube.com/watch?v=n8ep4leoN9A&feature=youtube_gdata_player (visited on 04/25/2014).

[181] Eric Brown. *Kit aims Snapdragon 800 at embedded Android designs*. HackerBoards. June 30, 2013. URL: http://hackerboards.com/snapdragon-800-dev-kit-targets-embedded-android-designs/ (visited on 11/04/2016).

[182] Peter Greenhalgh. *Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. Sept. 2011. URL: http://www.cl.cam.ac.uk/~rdm34/big.LITTLE.pdf (visited on 11/16/2016).