Measuring QoE of Interactive Workloads and Characterising Frequency Governors on Mobile Devices

Volker Seeker, Pavlos Petoumenos, Hugh Leather and Björn Franke
Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom
Email: v.g.seeker@sms.ed.ac.uk, ppetoume@inf.ed.ac.uk, hleather@inf.ed.ac.uk, bfranke@inf.ed.ac.uk

Abstract—**Mobile computing devices such as smartphones and tablets have become tightly integrated with many people's life, both at work and at home. Users spend large amounts of time interacting with their mobile device and demand an excellent user experience in terms of responsiveness, whilst simultaneously expecting a long battery life between charging cycles. Frequency governors, responsible for increasing or decreasing the CPU clock frequency depending on the current workload and external events, try to balance the two contrasting goals of high performance and low energy consumption. However, despite their critical role in providing energy efficiency it is difficult to measure the effectiveness of frequency governors in an interactive environment. In this paper we develop a novel methodology for creating repeatable, fully automated, realistic, workloads that can accurately measure time lag in interactive applications resulting from non-optimally selected operating frequencies. We also introduce a new metric capturing the user experience for different ANDROID frequency governors. We evaluate interactive workloads to demonstrate how our approach enables us to automatically record and replay sequences of user interactions for different system configurations. We demonstrate that none of the available ANDROID frequency governors performs particularly well, but leave substantial room for improvement. We show that energy savings of up to 27% are possible, whilst delivering a user experience that is better than that provided by the standard ANDROID frequency governor. We also show that it is possible to save 47% energy with performance that is indistinguishable from permanently running the CPU at the highest frequency.**

I. INTRODUCTION

Thanks to their immense functionality and portability mobile computing devices such as smartphones, tablets and ebook readers have become ubiquitous in the hands of consumers and corporate users alike. These gadgets make it possible for users to access email, browse the internet and download music and applications, regardless of location. While expecting a highly performant device and short system response times, users also want a long battery lifetime between charging cycles.

After the screen and radios, the CPU is one of the most energy consuming parts of a mobile system [1]. Several power saving strategies are used to find a good balance between energy consumption and performance. One of them is dynamic frequency and voltage scaling (DVFS) which in Linux is done by the frequency governor. This technique allows the OS to trade performance for power and energy, and visa 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. 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.

Our study shows, however, 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. 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 the user's point of view while evaluating how well a frequency governor performs to achieve the best energy efficiency while at the same time providing user satisfaction.

None of the current mobile benchmark suites, however, comes with an easy-to-use and deterministic method to evaluate user perception for an interactive workload [2,3,4]. A classic approach to evaluate user perception is using questionnaires [1,5]. 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 a tedious, but nearly impossible task, especially as the length of the benchmark exceeds a certain time span (in our study we include 24 hour workloads).

In this paper we introduce a methodology that allows us to record and replay custom interactive workloads on ANDROID mobile devices and automatically evaluate the effects of changes to the system in terms of user perception. Figure 1 shows the concept of our methodology. To get a clear picture of how the user perceives the system, we execute an interactive

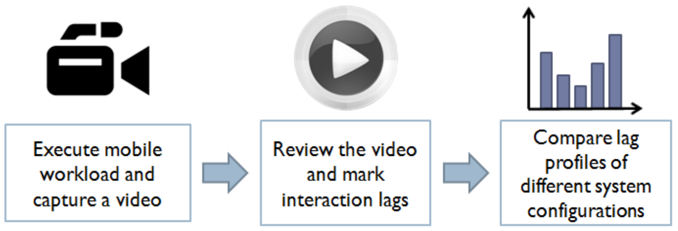
Execute mobile workload and capture a video
Review the video and mark interaction lags
Compare lag profiles of different system configurations

Fig. 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.

workload and capture a video of what the screen is showing. We then mark beginning and end of each *interaction lag* we find in the video. We call the time between user input and the time when the user feels the system has process his request *interaction lag* (see Figure 2). By marking begin and end of all interaction lags we create an interaction lag profile that lists the length of all lags the user perceived in the executed workload. We can then compare the durations of those lags to another execution of the same workload possibly using a different system configuration. From the interaction lag profiles of each captured video we derive a "*user irritation*" metric which allows us to make a decision about which system configuration was less irritating to the user due to shorter interaction lags.

To demonstrate the feasibility of our method and metric, we use them to investigate the potential of saving energy by improving the CPU frequency governor considering user perception. We compare the energy consumption and user irritation of three standard ANDROID frequency governors and find that neither of them performs particularly well. We derive optimal frequency profiles for each executed workload which show us that up to 27% energy savings are possible whilst maintaining a system performance that fully satisfies the user.

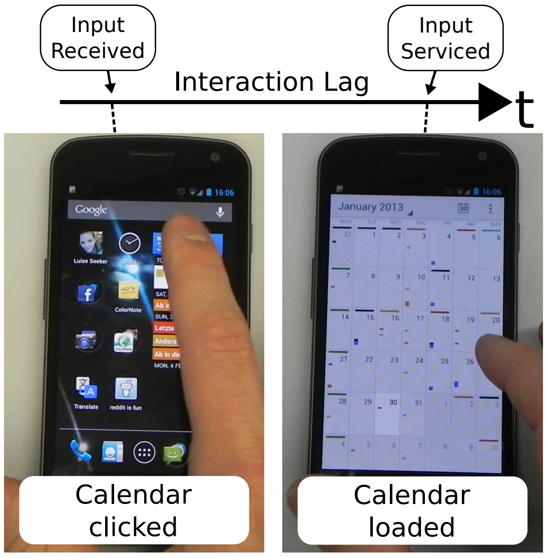

Fig. 2. Interaction lag is the time between user input and the time when the user feels the input has been serviced by the system. In the example, the interaction lag begins when the user clicks on the calendar and ends when the calendar is fully loaded.

A. Contributions

Among the contributions of this paper are:

1) a record and replay mechanism to deterministically replay realistic interactive workloads on the same/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
3) automatic detection of interaction lag, based on non-intrusive analysis of video output and device event queues,
4) a metric derived from interaction lag profiles to classify user irritation for a particular workload,
5) a study on how this measurement methodology can be use to develop a frequency governor with optimal energy-efficiency for interactive workloads. This is done whilst maintaining the same or even improving system responsiveness compared to three standard governors.

B. Motivating Example

Figure 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 different 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.

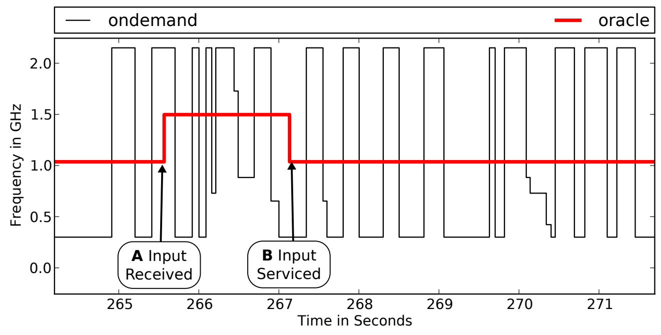

Fig. 3. Snapshot of the behavior of the *Ondemand* governor and another more energy efficient governor for one of this study's interactive workloads.

When we showed a video of this short example to a group of different users, they were not able to distinguish between the two frequency configurations, being fully 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. We identified three major issues 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.
3) *Ondemand* uses very low frequencies at times where a higher frequency would save energy due to the race-to-idle phenomenon [1]. This happens inside and outside of interaction lags.

What we need to know in order to avoid those issues is the user's point of view of the system. We need to know 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. We could then use this interaction lag information to rank a frequency governor in terms of energy efficiency and user satisfaction and later to construct an optimal one.

Since current mobile benchmark suites [2,3,4] do not offer a way of identifying user interaction lag, we set out to create a new methodology. Identifying the beginning and the end of user interactions for a mobile workload is a straightforward task. In our first experiments, we pointed a camera at an ANDROID Galaxy Nexus and started executing a workload. We opened the recorded video in a standard video editing tool and stepped through it frame by frame (see step two in Figure 1). Every time we identified a frame as the one where the user submitted an input command, we set a *begin-marker*. Every time we decided that the system now looks like it has serviced the input, we set an *end-marker*. Afterwards, we extracted the number of frames between all markers and had our interaction lag profile. We could now compare two profiles of different executions of the same workload or overlay them with the corresponding frequency profile to see what the governor did.

Unfortunately, the process of marking up a video showing only 10 minutes (18000 frames at 30 fps) of relatively interaction intensive workload takes already about 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 we used this process to produce enough data for a thorough study. As an example, the results presented in Section IV required 5 different 10-minute workloads, each one executed for 17 different frequency configurations, with each configuration run 5 times in order to get statistically sound data. That translates into 4250 minutes of video material, for which we would need 1800 hours to markup, or almost a whole man-year. It is clear that if we want to be able to capture and study interactive workloads, we need to automate this process to a high degree. In the next section we will present such an automated novel methodology which allows us to reduce the manual work to a total time of 40 minutes which is a factor of 2700x.

A second important requirement for improving the frequency governor's energy efficiency is to have realistic and repeatable workloads. We asked a group of different users to execute the legacy mobile benchmark suite as proposed in [2]. 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. We found 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 analyze 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. We need a way of recording and replaying actual, rather than artificial, user interactions with millisecond accuracy to get representative workloads. These need to be repeatable without major deviations in order to compare multiple executions. With our technique, users can create repeatable and realistic workloads as they would naturally execute them.

C. Overview

This paper is structured as follows. In Section II we present our novel approach of automatically detecting interaction lag in realistic and repeatable workloads and how we derive a user irritation metric. This is followed in Section III by a description of our experimental setup to apply our methodology to a frequency governor study. A presentation of experimental results can be found in Section IV. We discuss related work in Section V before we summarise and conclude in Section VI.

II. METHODOLOGY

In this section we will describe in detail how our methodology works. We start by giving an overview about the automation steps we did and then explain them in detail.

A. Automation Steps Overview

Instead of executing a workload manually for each run, we record and replay user inputs. We capture input events directly from the Linux input subsystem, so we are able to replay them in exactly the same way and with accurate timings whenever needed. We do that by following the same approach as presented in other studies [6].

Knowing the exact timings for all input events, already gives us the beginning of each interaction lag. Now we also automated finding the ending of an interaction lag. The ending is the time when the user feels that the system has serviced his input. To do this we implemented a matcher algorithm 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 video we captured from 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.

Now we create the image database which we call *annotating the workload*. Annotating a workload means selecting an image for each interaction lag that shows how the mobile screen looks when the 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. We made the process

[1]Servicing the input quickly using a higher frequency and putting the CPU to a low power idle state might be preferable to servicing the input using a lower frequency.

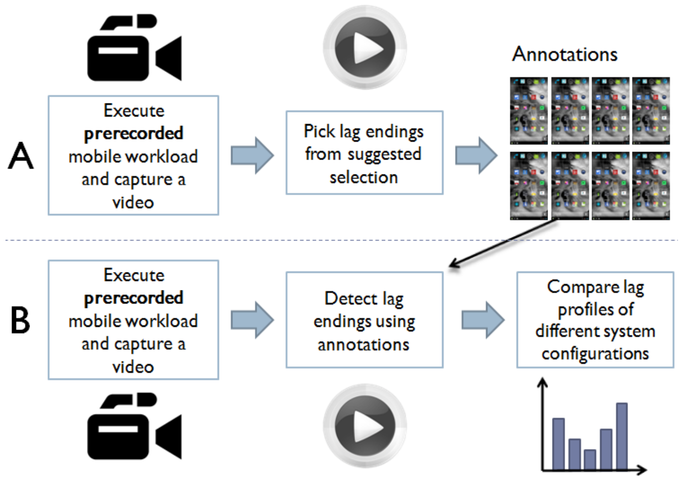
Annotations
A
Execute **prerecorded** mobile workload and capture a video
Pick lag endings from suggested selection
B
Execute **prerecorded** mobile workload and capture a video
Detect lag endings using annotations
Compare lag profiles of different system configurations

Fig. 4. This figure shows the automated version of our 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 we use the annotation database to automatically mark up a video of the workload's execution and produce a lag profile.

easy for the workload creator by automating most of it as well. In our manual markup method in Section I the user 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 user 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 user only needs to pick the right one. This takes in average only a couple of seconds per interaction lag. The image the user picked is then added to the workload's image database which is later used by the matcher.

Figure 4 shows the automated version of our method derived from the former concept in Figure 1. Part A shows the annotation step and needs to be executed only once. Here we run a prerecorded workload and capture a video of it. Our suggester algorithm then presents a selection of potential lag ending frames for each lag beginning and the user 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.

B. Automatic Record and Replay of Interactive Workloads

In order to accurately record and replay a workload the user executed on the mobile device, we capture input events directly from the Linux input subsystem. This system provides a standard interface for handling the input provided by various peripheral devices and sensors. On a mobile device that would be for example touch screen, hardware buttons, light sensor, etc. The hardware version of the input events captured by the single device drivers are 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*. A single touch is composed out of multiple input events as shown in Figure 5.

```
/dev/input/event1: 0003 0039 00000003
/dev/input/event1: 0003 0030 0000000e
/dev/input/event1: 0003 003a 00000089
/dev/input/event1: 0003 0035 0000016b
/dev/input/event1: 0003 0036 000001a3
/dev/input/event1: 0000 0000 00000000
/dev/input/event1: 0003 0039 ffffffff
/dev/input/event1: 0000 0000 00000000
```

Fig. 5. An example of the GETEVENT input recording.

The first hexadecimal 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.

1) Record: ANDROID provides a tool called GETEVENT which is a front-end to reading the */dev* input event interface. When we record a workload, we use this tool to capture executed input events together with exact timestamps. The recording process needs no external hardware support, it is executed on the user's device, while it is carried with them about their daily business.

2) 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 our recorded event trace accurately. Therefore, we implemented our own event replay agent. This agent knows the input event trace we recorded and replays it with accurate timings.

C. Capturing Screen Output

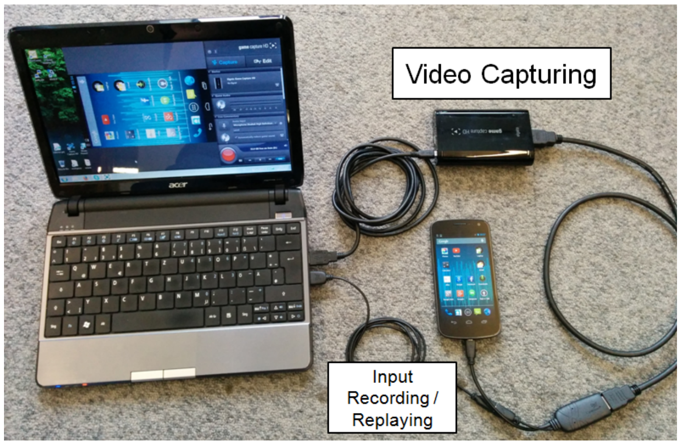

Fig. 6. We capture a video of the mobile screen output by recording an HDMI signal with a video capture device like the *Elgato Game Capture HD* [7].

Figure 6 shows how we capture a video of the mobile device screen. Rather than using a camera, we now capture the direct screen output via HDMI. This way we avoid image artifacts which would significantly complicate the process of

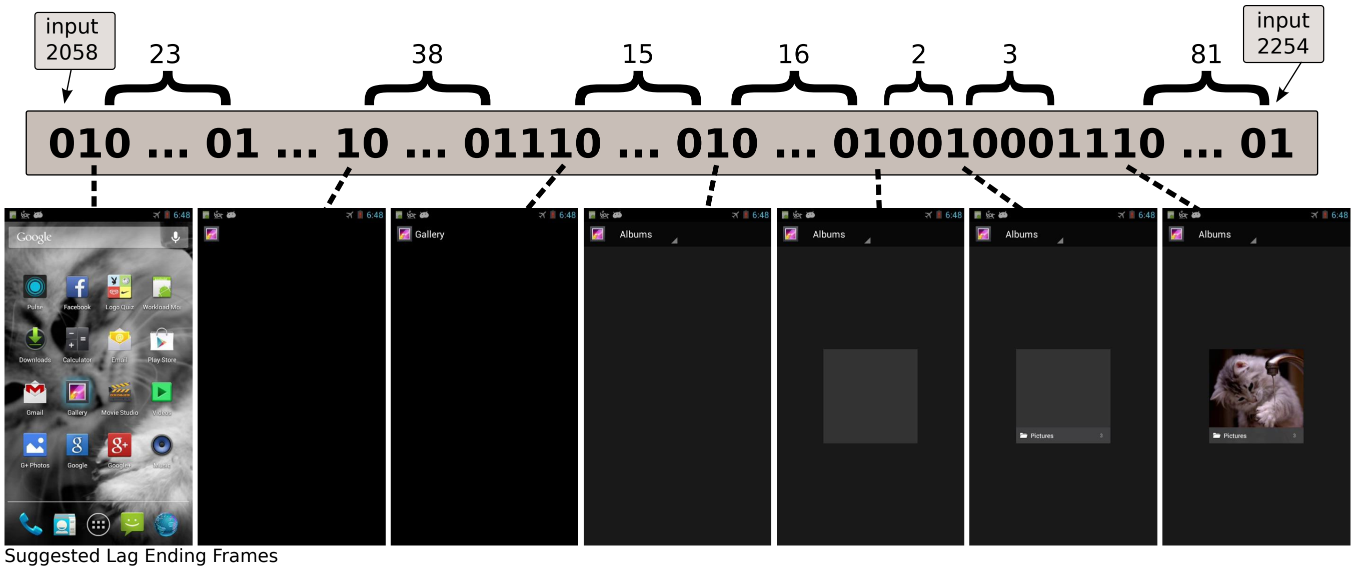

Fig. 7. The suggester algorithm maps successive video frames to a sequence of ones and zeros. A zero is assigned to a frame that looks equal to its predecessor and a one to each frame that differs from it. Each one preceding a zero is then suggested as potential lag ending since it marks the beginning of a period of still standing images.

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* [7] 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.

D. Semi-Automatic Markup of Workload Videos

As mentioned in section II-A we are using a semi-automatic process of 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 user then only needs to pick the correct one for each lag.

Figure 7 shows an example of how our 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 images on the bottom of Figure 7 are all suggestions made by the algorithm while the ones and zeros in the long box above show the suggester's inner representation of the video frames. The small box on the left side shows that an input occurs at video frame 2058 and the small box on the right shows the next input at frame 2254. The curly brackets summarize chains of zeros.

The point at which users determine the end of processing an input is always the last of some number of changing frames. The end point is never during a period of unchanging frames. 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. The algorithm then suggests 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 we pick 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 7 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 user has to look at is therefore reduced by a factor of 20. When a workload contains long periods without screen updates (as is the case with our 24 hour workload), 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 our example, the number suggestions would be reduced to 2 and we would still safely catch the correct one. Our workload creation GUI allows these settings to be explored and tuned easily.

E. Detecting Lag Endings Using the Annotation Database

Now that we have produced an annotation database containing an image of how each interaction lag ending is expected to look like, we can use it to mark up videos of any further execution of the same workload. Our 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

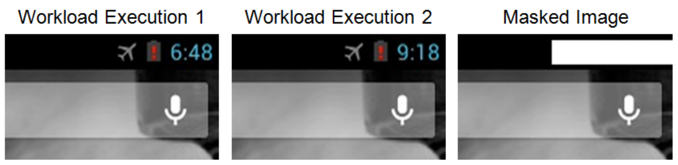

Fig. 8. We can mask out parts of the images being compared to handle a certain degree of non determinism between workload executions. In the example, we mask out the clock so the matcher can find the required ending image for different workload executions.

the annotation data base 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. We try to avoid these contents in our workloads, we are, however, 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 some 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 8). 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 send. The suggested lag ending therefore looks like the beginning. In this case the user can specify that the matcher should look for the second occurrence of the required image. Our GUI makes it easy for users 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 analyzed fully automatically.

F. User Irritation Metric

Since the inputs are always the same for different executions of the same workload, there will always be the same number of interaction lags. Our method produces a lag profile after evaluating a video which lists the lag length for each interaction lag in the evaluated video. In order to compare lag profiles of different executions of the same workload in terms of user irritation, we introduce a new metric.

Figure 9 shows a timeline for a single interaction lag. The beginning is marked and each circled number stands for a lag ending. The endings were found in experiments 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. The user then sets an *Irritation Threshold*. If the lag length is below this threshold, it does not count as

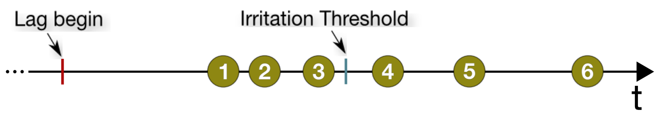

Fig. 9. This figure shows the timeline of a single interaction lag. Each circled number stands for the lag ending of a specific system configuration. Each lag length that stays below the specified Irritation Threshold does count as not irritating and for each lag length that exceeds it a penalty is applied.

irritating to the user, if it is above, we give an irritation penalty. The penalty is the amount of time the lag duration is above the threshold. Our metric is an accumulation of the penalty for each lag in the workload and therefore the total amount of time a user is irritated by too long lag times in a certain workload.

In our method, the Irritation Threshold is set independently for each lag. When picking the interaction lag ending from the suggested selection, the user can choose the threshold from a standard HCI model [8] which offers four interaction categories: typing (150ms), simple frequent task (1s), common task (4s) and complex task (12s). He can also apply a custom model or specify each Irritation Threshold individually. Our GUI makes custom thresholds easy to apply for each lag or use a standard HCI model or other common settings.

III. Experimental Setup and Execution

In the following sections we demonstrate the feasibility of our method. We analyze the three standard ANDROID frequency governors in terms of energy consumption and user irritation. We will also derive an optimal frequency profile for each executed workload which shows how much more energy savings are possible whilst maintaining a system performance that fully satisfies the user.

A. Workloads

TABLE I. A rough description of the main activities the users were executing in each workload.

Dataset	Description
01	Image manipulation with Gallery application.
02	Logo Quiz game.
03	Pulse News widget and multimedia text messaging.
04	Movie Studio video creation.
05	Pulse News application.

To create our workloads we asked five people to use a mobile device with our recording system installed for ten minutes each. No further instructions were given, beyond asking that they "exercise the software". Their interactions were recorded on the device (see Table I). Before getting access to the device it was reset to a known state to ensure that the recorded workloads could be rerun from that same state later. Among the applications we had pre-installed and made use of by our volunteers were Facebook, Pulse News, a Logo Quiz game, the Play Store, the Gallery, Gmail, the Music Player, the Calculator. In addition, to demonstrate the capabilities of our system, one user recorded a workload for a full timespan of 24 hours.

In total we recorded 5 different workloads. We replay each of them for each available core frequency. During those executions the frequency is fixed for the whole runtime. The Snapdragon 8074 processor we use allows 14 different frequency points. We also replayed each workload for each of the three governors. To reduce the statistical error, we repeat this process 5 times per workload. Altogether we execute each workload $5 * (14 + 3) = 85$ times.

B. Frequency Governors and Oracle

The three frequency governors we look at are the *Ondemand*, the *Conservative* and the *Interactive* governor. *Ondemand* and *Conservative* are included in almost every modern Linux-based system 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.

When executing a workload we collect frequency and CPU load traces in the background for each run. We then use the traces of all fixed frequency workload executions to compose an optimal frequency trace (oracle) that uses the least amount of energy possible without irritating the user. Consider the interaction lag example in Figure 9 where each circled 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 we pick the lowest frequency and corresponding load for each lag that is still below the chosen irritation threshold. In Figure 9 that would be frequency ③. For each lag we set the irritation threshold to 110% of what the fastest frequency could achieve. We assume that the user does not notice a 10% difference between lag timings[2]. For each interval in a workload where there is no lag, we pick the frequency and corresponding load that had the lowest overall energy consumption for the complete workload. Figure 3 shows for a short timespan how the oracle's frequency selection compared to a governor looks like.

We calculate energy consumption for each frequency-load profile of the governors and the oracle by using a power model and compare them with each other. Additionally, we derive our user irritation metric from their interaction lag profiles generated by our methodology and compare those too.

C. Experiment Platform

For our frequency governor experiment we are using the Qualcomm Dragonboard APQ8074 which is based on the Snapdragon 8074 quad core processor. It has the same underlying architecture as the Google Nexus 5 but allows us easier access to connectors and interfaces to measure energy and modify various parts of the system. We run ANDROID Jelly Bean version 4.2.2 with kernel 3.4.0. For our experiments we switch off all cores except one while evaluating frequency governors to reduce statistical noise from load balancing between different cores.

To calculate energy for our oracle and the governors using the collected frequency and load profiles, we create a power model for the Snapdragon processor. We execute a CPU intensive micro benchmark for each core frequency and measure overall system power. We then subtract the idle system power to get dynamic core power for each frequency.

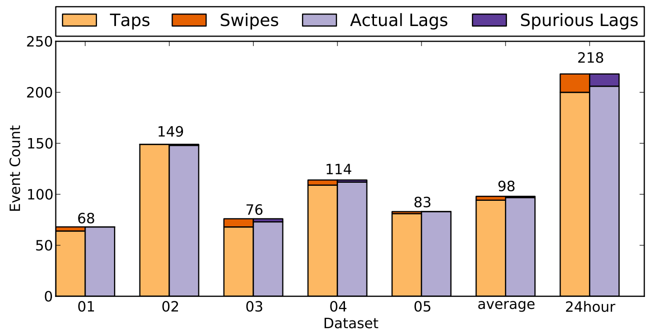

Fig. 10. The graph shows an input classification for all workloads and the 24 hour workload. For each pair of bars, the left bar shows tap inputs and swipe inputs, while the right bars shows the lags and spurious lags.

IV. DISCUSSION OF RESULTS

Figure 10 shows an input classification for all datasets we used for our evaluation including the 24 hour workload. The left bars show tap inputs in a light color and swipe inputs in a darker color. The tap inputs are dominating due to the nature of our workloads. The right bars show the number of inputs we identified as actual lags in a light color and the inputs that were spurious lags in a darker color. It can happen that an input event does not lead to any 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, we consider those inputs as spurious lags and ignore them as well. The 10 minute datasets we use for our frequency governor evaluation are interaction intensive as the graphs demonstrate when compared to the 24 hour workload.

Figure 11 shows violin plots of lag durations for each frequency configuration for a single dataset with a zoomed in version on the right side. Again we present only the findings for Dataset 01 due to their similarity. The left graph has a single kernel plot for the *Ondemand* frequency governor in the top right corner. The kernel plot shows that with an average of about 500ms, most of the lags are rather short. There is the occasional very long lag with up to 13 or 12 seconds in the lowest frequency. In this particular workload we were editing images and saving the results to the sd card. These long durations occur since we consider the whole time the image needs to be saved as a lag. The lag durations are in all workloads significantly longer for the *Conservative* governor while *Interactive* and *Ondemand* are close to each other. The frequency configurations with a fixed frequency settle on an average lag length the higher the frequency gets. This already indicates that we can reach the same user perceived system response time for most of the interactions with a medium frequency as we would with the the highest.

In Figure 12 we show our user irritation metric on the left and energy consumption with the oracle as baseline on the right for Dataset 02. The smaller graph in the top right corner of the left graph shows a zoom in on user irritation for governors only. It is clearly visible how the user irritation shrinks quickly the faster the frequency becomes. The shape

[2]Customized irritation thresholds are easily specified in our workload construction GUI.

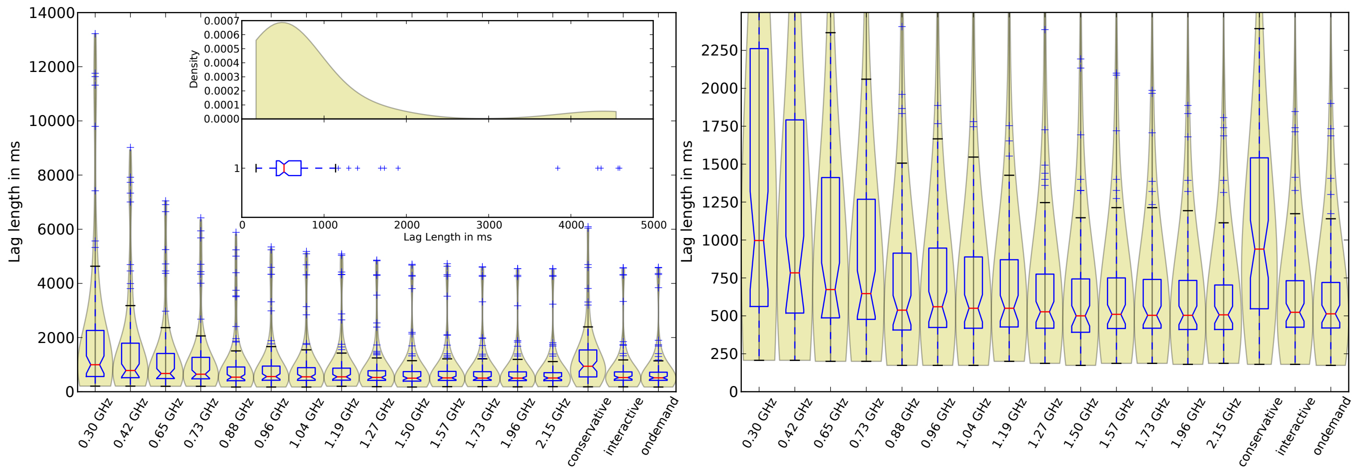
Fig. 11. Violin plots of the lag durations for all frequency configurations for Dataset 01 with a zoomed in version on the right. Boxes extend from lower to upper quartile values, with a line at the median. The whiskers show the range of the lag length at 1.5 IRQ, while flier points are those past the end of the whiskers. In the top right corner of the left graph is single kernel plot which shows the distribution of lag durations for only the *Ondemand* governor.

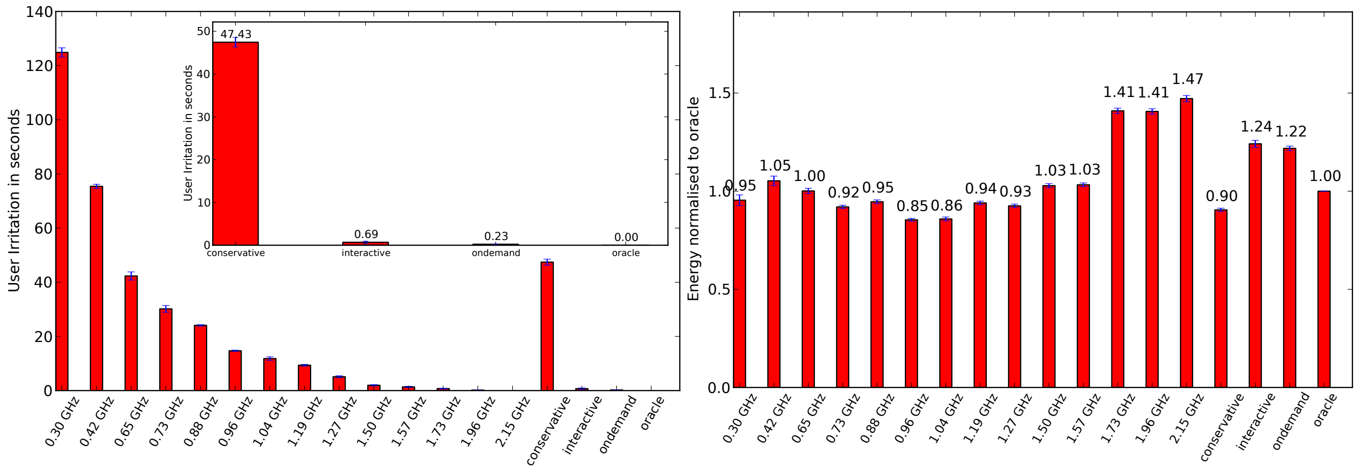
Fig. 12. The left graph shows our user irritation metric for all frequency configurations of Dataset 02 with a governor only version in the top right corner. The right graph shows the energy consumption for all frequency configurations of the same dataset. All energy values are normalized to the oracle's energy consumption on the far right.

of the bars is very similar to the violin plots in Figure 11. By definition, both the oracle and the fastest frequency are not irritating to the user at all. As was already apparent from the lag durations, the *Conservative* governor is significantly more irritating with longer lag durations than *Interactive* and *Ondemand* which are again close together. In fact, the latter two governors are doing a good job in terms of user irritation since they are for the whole workload less than 1 second above our oracle.

The energy consumption does not follow the same shape as irritation and lag durations. It varies due to the race-to-idle phenomenon which means that in some cases less energy is consumed if a task is executed in a short time using a high frequency than in a longer time with a lower frequency. The most energy efficient frequency for the workload shown in Figure 12 is 0.96 GHz where the balance between task execution time and low power consumption reaches an optimum. This is also the frequency we would pick for all non lag periods to construct the oracle for this workload. All energy values are normalized to the oracle's energy consumption on the far right. *Interactive* and *Ondemand* need significantly more energy than the *Conservative* governor with up to 24% above the oracle. The *Conservative* governor, however, performs 10% better than the oracle.

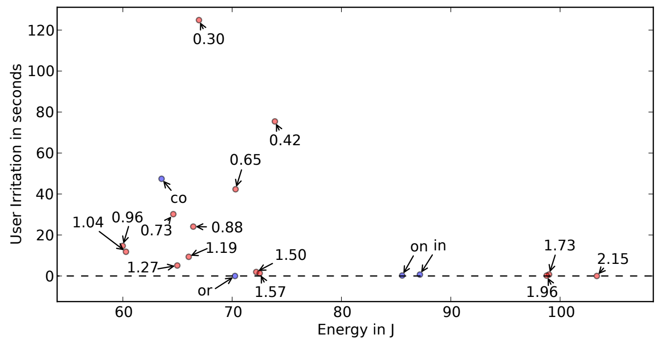
Fig. 13. Scatterplot of energy and irritation metric for Dataset 02 with frequency governors in blue and fixed frequencies in red. Oracle and the fastest frequency are laying on the base line for user irritation.

Figure 13 shows a scatter plot of the irritation and energy metrics with the total energy consumption on the x-axis and the user irritation in seconds on the y-axis. We chose a blue dot for governors and a red one for fixed frequencies. While *Interactive* and *Ondemand* are close to each other and the oracle base line in terms of user irritation, they still leave a lot of room for improving their energy consumption. The

Conservative governor needs less energy but is far more irritating to the user than our proposed oracle. In this graph it is also interesting to see that for this particular workload a fixed frequency like 1.50 GHz or 1.57 GHz would have done a better job than all standard governors while being only slightly more irritating to the user than our oracle.

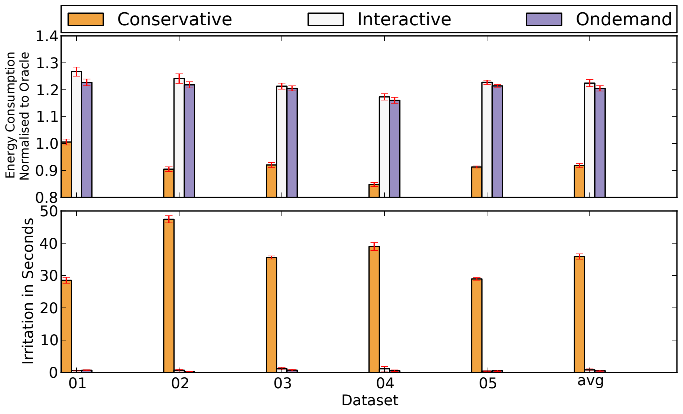

Fig. 14. This figure shows a summary of the governor's energy consumption for all datasets in the top graph. The energy values are normalised to the oracle. The bottom graph shows a summary of the governor's user irritation for all datasets.

Figure 14 shows a summary of energy and irritation of the governors for all datasets we used in this study. The top graph is showing energy and all values are normalised to the corresponding oracle's energy consumption. The *Conservative* governor's energy consumption is on average 8% better than the oracle. *Interactive* and *Ondemand* need on average 22% and 20% more energy. The graph on the bottom shows a summary of user irritation for all datasets. *Conservative* is significantly more irritating than *Interactive* and *Ondemand* and needs on average 36 seconds longer for all lags together. The latter two, however, are close to our oracle and need on average only about 1 second more for all lags in a workload together.

V. Related Work

A. Interactive Mobile Workloads

Gutierrez et al. [2] 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.* [4] 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.* [3] also present a mobile benchmark suite comprising video playback, image rendering and 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. *Gomez et al.* [6] is probably most similar to our approach as their system supports record and replay of user input events. Using their tool they are capable of reproducing bugs in popular ANDROID applications, but their study does not focus on user perception. *Sunwoo et al.* [9] 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 synchronizes service of input by evaluating the frame buffer. However, as with other studies this does not consider HCI based user satisfaction metrics. GOOGLE MONKEY [10] is a stress tester for ANDROID applications and generates random input. ROBOTIUM [11], and GOOGLE's MONKEYRUNNER [12], are automation tools tied to the ANDROID OS. ROBOTIUM is additionally tied to a particular application and requires the user to write an automation script using knowledge of the application layout and the ANDROID API. MONKEYRUNNER is more general and can replay mouse and keyboard events rather than application-specific events, but fires input events at fixed times rather than adaptively and synchronized with the application. Manual development of automation files for ROBOTIUM and MONKEYRUNNER is a rather tedious process. GUITAR [13] extends ANDROID SDK's MONKEYRUNNER tool to allow users to create their own test cases with a point-and-click interface that captures press events. However, GUITAR does not support touchscreen gestures, e.g. swipe and zoom, or other input devices, e.g. accelerometer and compass. XNEE [14] is an automation tool for LINUX, which records user input and related X 11 events events is specific to the software stack running on top of the LINUX kernel and not applicable to ANDROID.

B. *User Perception Based* DVFS

Shye et al. [15] 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 system, 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.* [16]. Experiments are conducted with real user to investigate how different frequency levels in different scenarios affect biometric input. Again, workloads are not replayable and mainly involve desktop applications. *Mallik et al.* [17] use the same approach to evaluate user satisfaction based on visual output. They measure the rate of pixel intensity changed over time and use this metric to to control frequency governor. Similar to other studies, they use questionnaires, focus on DVFS for desktop platforms and do not make provising for replaying workloads with modified settings to evaluate success. *Shye et al.* [1] 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. *Yan et al.* [18] aim to improve DVFS based on user perceived latencies in system response time for interactive workloads. It monitors events in the LINUX X window system to measure latency and to control the frequency governor. The focus of this work is on desktop systems. *Lorch et al.* [19] analyze traces of user interface events to derive a heuristic to determine when user interface task completes, which is subsequently used to influence DVFS decisions based on abstract user satisfaction thresholds.

VI. SUMMARY AND CONCLUSIONS

In this paper we evaluated the energy saving potential of current ANDROID frequency governors when considering user perception. In order to do that we introduced a novel semi-automatic methodology to identify interaction lag in interactive mobile workloads. By recording and replaying user interactions with an ANDROID device we could build and use real interactive workloads for our experiments. We also presented a metric to quantify user irritation for a workload by evaluating the interaction lag data produced by our method and setting maximum deadlines for each interaction. We demonstrated the feasibility of our method and metric by evaluating three standard ANDROID governors. For each workload we generated an optimal oracle profile 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 our predicted optimal profile we found that *Interactive* and *Ondemand* leave room for more energy savings with up to 27% while the *Conservative* governor needs on average 8% less energy than our oracle. It shows, however, a significantly higher user irritation. On average the user is irritated for 36 seconds over the course of a 10 minute workload. *Interactive* and *Ondemand* need on average 22% and 20% more energy but are a lot closer to our optimum in terms of irritation (on average less than 1 second). In this study we could successfully show the usefulness of our method for frequency governors but it can also easily be applied to other system properties. Our technique can be used to evaluate operating system, compiler or architectural changes for the first time with repeatable and realistic workloads considering user perceptions.

In our future work we plan to further extend our methodology. In its current state we are not considering networking workloads since they are heavily non deterministic. If the user, for example, starts the browser and opens a news web page, it might look completely different between different workload executions. One could circumvent this problem by using a workload aware network proxy that creates a deterministic environment for network accesses. We also plan to include workloads that are dominated by *Jank* [20] type lags where 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. We also plan to integrate our proposed user irritation metric into the ANDROID display stack in order to make energy efficient frequency governor decisions at runtime.

This work is funded under the EPSRC *grant,* ALEA *(EP/H044752/1).*

REFERENCES

[1] A. Shye, B. Scholbrock, and G. 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*, ser. MICRO 42. ACM, pp. 168–178.

[2] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 81–90.

[3] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - MobileBench," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 133–142.

[4] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators." [Online]. Available: http://asg.ict.ac.cn/projects/moby/downloads/Moby-ISPASS2014.pdf

[5] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay, "MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones." ACM, pp. 57–70.

[6] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: timing- and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 72–81.

[7] Game capture HD. [Online]. Available: http://www.elgato.com/en/gaming/game-capture-hd

[8] B. Schneiderman, "Designing the user interface: strategies for effective human-computer interaction."

[9] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. Emmons, and N. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 113–122.

[10] Google MonkeyRunner. [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html

[11] Robotium - the world's leading android test automation framework. [Online]. Available: http://code.google.com/p/robotium/

[12] Google UI/Application exerciser monkey. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[13] GUITAR - a GUI testing framework. [Online]. Available: http://sourceforge.net/projects/guitar/

[14] Xnee. [Online]. Available: http://www.gnu.org/software/xnee/

[15] A. Shye, B. Ozisikyilmaz, A. Mallik, G. Memik, P. Dinda, R. Dick, and A. Choudhary, "Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction," in *35th International Symposium on Computer Architecture, 2008. ISCA '08*, pp. 427–438.

[16] A. Shye, Y. Pan, B. Scholbrock, J. Miller, G. Memik, P. Dinda, and R. Dick, "Power to the people: Leveraging human physiological traits to control microprocessor frequency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41*, pp. 188–199.

[17] A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda, "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*, ser. ASPLOS XIII. ACM, pp. 70–79.

[18] L. Yan, L. Zhong, and N. Jha, "User-perceived latency driven voltage scaling for interactive applications," in *Design Automation Conference, 2005. Proceedings. 42nd*, pp. 624–627.

[19] J. Lorch and A. Smith, "Using user interface event information in dynamic voltage scaling algorithms," in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003*, pp. 46–55.

[20] N. Duca and T. Wiltzius, "Google I/O 2013 - jank free: Chrome rendering performance." [Online]. Available: http://www.youtube.com/watch?v=n8ep4leoN9A&feature=youtube_gdata_player