Green smartphone GPUs: Optimizing energy consumption using GPUFreq scaling governors

Green Smartphone GPUs: Optimizing Energy Consumption using GPUFreq Scaling Governors

Enas Ahmad and Basem Shihada

Computer, Electrical and Mathematical Science Engineering Division
King Abdullah University of Science and Technology (KAUST)
{enas.ahmad, basem.shihada}@kaust.edu.sa

***Abstract*—Modern smartphones are limited by their short battery life. The advancement of the graphical performance is considered as one of the main reasons behind the massive battery drainage in smartphones. In this paper we present a novel implementation of the GPUFreq Scaling Governors, a Dynamic Voltage and Frequency Scaling (DVFS) model implemented in the Android Linux kernel for dynamically scaling smartphone Graphical Processing Units (GPUs). The GPUFreq governors offer users multiple variations and alternatives in controlling the power consumption and performance of their GPUs. We implemented and evaluated our model on a smartphone GPU and measured the energy performance using an external power monitor. The results show that the energy consumption of smartphone GPUs can be significantly reduced with a minor effect on the GPU performance.**

***Index Terms*—Smartphones, GPU, DVFS, CPUFreq Governors, Energy Efficiency**

I. Introduction

The graphical capabilities of mobile devices have advanced tremendously over the past few years. This leap of improvement was only made possible by introducing Graphical Processing Unit (GPU) chipsets to mobile phones. Instead of relying on the CPU to process its graphical content, modern phones use specialized powerful circuits dedicated to process any graphical data much more efficiently. NVIDIA [1], predicts that soon enough smartphones will even compete with game consoles in their graphical performance. However, mobile phones will still suffer from several limitations such as size, weight, and most importantly battery power.

A few years back, most mobile phones were able to last three to four days without requiring a recharge. However, users today complain about the short battery life of their smartphones, requiring a daily recharge, if not more often. According to a study done by [2], 80% of mobile phone users are searching for various measures to increase their battery lifetime. Thus, designing energy-efficient smartphones while keeping up with their newer capabilities is now becoming vital.

Rich graphical interface components and mobile games continuously push the performance boundaries of smartphone GPUs. Consequently, smartphone vendors increasingly provision their phones with more powerful GPUs to ensure smooth rendering of graphics, which in its turn increases the energy consumption of the phone. Operating systems such as Android are now fully hardware accelerated [3]. Every drawing operation and rendering of both 2D and 3D graphics are carried out solely by the GPU. Moreover, smartphones nowadays are equipped with large screens with very high resolution. That facilitates processing of large, complex graphical data, leading to more load on the GPU. All of these factors that effect the energy consumption are considered inevitable, since they are either related to the underlying hardware, or the natural progress of mobile graphics. Therefore, one solution to overcome the poor battery performance is to make smartphone GPUs more energy-efficient by enabling them to handle the increasing load with minimum energy consumption. One way to achieve this is by implementing GPU scaling.

Smartphone applications can vary from simple 2D web browsers or text based editors, to high resolution 3D gaming. This large variation in rendered graphics encourages using dynamic scaling techniques for the GPU based on its real time utilization. Energy can potentially be saved by dynamically adjusting any processor's frequency and voltage to the current usage instead of running it at its highest performance level the entire time. The Linux kernel utilizes this methodology for the CPU by defining what is famously known as "CPUFreq Scaling Governors" [4]. Each of these governors offers a different scaling algorithm for the CPU with a trade-off between performance and power. The main Linux governors are: *Performance*, *Powersave*, *Userspace*, *On-demand*, and *Conservative*. The CPUFreq governors of Linux have shown to be one of the most effective methods to reduce energy consumption and heat emissions of computer systems. In this paper, we explore the feasibility of establishing a similar model for smartphone GPUs.

The power efficiency of mobile GPUs has been studied in prior work (details in Section II). However, these studies were of a simulation-based and haven't been evaluated or implemented on real mobile GPUs. Moreover, the power measurements of these studies are only approximations derived from theoretical energy models. We believe that simulating smartphone GPUs is not sufficient to reflect the variations, characteristics, and constraints of a real smartphone environment. In this work, we practically address the energy consumption of smartphone GPUs, and propose a novel implementation of the GPUFreq Scaling Governors as a part of the Android Linux kernel.

Our model allows users to select one of four governors

in order to control the scaling of their phones' GPU. The aim is to enhance the power efficiency of smartphone GPUs, in addition to providing users with similar variations and alternatives offered to them by the CPUFreq governors. Unlike previous work in this area, we implemented and evaluated our approach on a modern smartphone GPU, and acquired actual energy measurements using an external power monitor. We show through extensive experimental the feasibility of inheriting the Linux scaling governors into smartphone GPUs.

The rest of the paper is organized as follows: Section II presents some related work; Section III explains the concept of DVFS; Section IV discusses the Linux CPUFreq Governors; Section V gives an overview of mobile GPU architecture. Section VI presents the design and implementation of our new model of GPUFreq Scaling Governors; Section 7 displays our experimental setup; Section VIII presents the evaluation results; and finally Section IX concludes our work.

II. Related Work

Due to the limited battery power of smartphones, the design of their GPUs should consider the energy consumption as a main factor even prior to the performance [5]. Some recent studies have looked into the energy efficiency of GPUs in mobile devices. Authors in [6] analyze the trade-off between energy and arithmetic precision of mobile graphics processor, by focusing on the vertex transformation stage in the graphics pipeline. The goal is to measure the energy savings that can be achieved by lowering the accuracy of arithmetic operations to an acceptable level. Their energy model is based on approximating the energy usage of the number of signal transitions per operation. Using the ATTILA GPU simulation framework [7], their results show that around 23% of the energy can be saved by lowering the precision of the arithmetic operations, while maintaining a good quality of the rendered image.

DVFS is one of the traditional power optimization techniques for general processors. Thus, researchers investigated the applicability of this method on GPUs. A primary study in [8] shows that 3D interactive gaming is amendable to DVFS. They use the open source Quake II game engine [9] for calculating their measurements, proving that 3D games have variations in their frames processing workload. Such variation enables the use of DVFS, which they predict can save significant amount of energy. Authors in [10] built their approach upon this conclusion. They present a quantitative study of the power consumption of mobile 3D games using three embedded processors to simulate the different stages of a mobile graphics pipeline. For measuring the power consumption, they employ an instruction level energy model, which is based on measurements of the current drawn by a commercial processor. The trace-driven simulation approach they used is implemented by modifying the OpenGL\ES library adding trace triggers. Their measurements confirm the existence of an imbalance of workload between different graphics stages and applications. They evaluate 6 DVFS schemes each with a diffident workload prediction algorithm. The results show that applying DVFS to graphics pipeline saves up to 50% of energy. However, their energy model is derived from a general processor rather than a graphics processor, which differ in their architecture, complexity, thus their power consumption.

Similar work in [11] focuses on enhancing the workload prediction for tile-base architecture GPUs. They propose two DVFS schemes, the first based on *tile-history* prediction, and the second based on *tile-rank* prediction. To evaluate the efficiency of the two approaches, they modify the ATTILA GPU simulation framework emulating a tile-based mobile graphics architecture. Their results show that both schemes save around 58% of energy consumption, with higher quality achieved by the *tile-rank* prediction. But unlike the previously mentioned studies, the method for driving their power model measurements was not clearly stated.

III. Dynamic Voltage and Frequency Scaling

DVFS is one of the commonly-used techniques for power saving in electronic devices. The power consumed by any Complementary Metal-Oxide-Semiconductor (CMOS) component is proportional to the voltage and frequency [12], as shown by

$$Power \propto Voltage^2 \times Frequency$$

Thus, when the frequency is lowered, the consumed power is decreases proportionally, while lowering the voltage causes the power consumption to drop quadratically [13]. DVFS takes the advantage of a common feature across a large number of computer applications: the average computational throughput is often much less than the computational peek capacity of the processor [14]. Therefore, a significant amount of power can be saved by scaling the frequency and voltage of the processor according to the real-time load of the running applications. This power reduction, in turn, reduces the heat emission in large scale supercomputers, and saves battery in low-scale embedded processors. As mentioned previously, in addition to CPUs many studies have shown the applicability of this method in GPUs as well. These findings encourage the study of the effectiveness of different DVFS schemes on real smartphone GPUs. Prior to presenting the design and implementation of our model, in the next section we introduce the Linux CPUFreq governors which are the inspiration behind this work.

IV. Linux CPUFreq Governors

CPU drivers set the CPU frequency to a single value. Therefore, in order to implement dynamic scaling, an additional software layer is needed to inform these drivers of which frequency to apply at run time. In Linux, the policies for dynamically controlling the CPU frequency and voltage are defined in the CPUFreq Governors' layer [4]. Users can select the best governor policy that fits their usage patterns. Since Android is based on Linux, it has also inherited these scaling governors. Figure 1 explains the flow of the CPUFreq governors control.

Currently, there are five main official governors in the Linux kernel:

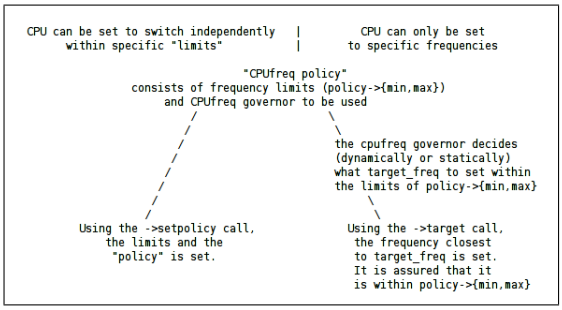


Fig. 1. CPUFreq governors control flow [4]

Algorithm 1: *On-demand* CPUFreq Governor

```
1 for every sampling_rate do
2     if cpu_load ≥ up_threshold then
3         target_freq=max_load_freq
4     else
5         if cpu_load ≺ up_threshold-down_differential then
6             target_freq= max_load_freq /(up_threshold-down_differential)
7         end
8     end
9 end
```

- *Performance*: The *Performance* governor statically sets the frequency to the highest value within the minimum and maximum limits of the frequency policy. This governor aims to maximize system performance, regardless of the energy consumption. Thus, it is mainly used for benchmarking purposes.
- *Powersave*: The *Powersave* governor is the opposite of the *Performance* governor, it statically sets the frequency to the lowest value within the minimum and maximum limits of the frequency policy. The governor aims to achieve the lowest energy consumption possible.
- *Usersapce*: The *Userspace* governor has no defined algorithm implementation. Rather, it hands out the control to any rooted user to set the value of the CPU frequency. This is achieved by making the *sysfs* file *"scaling_setspeed"* accessible in the user-space.
- *On-demand*: The *On-demand* governor, as defined in Algorithm 1, sets the CPU frequency dynamically based on the current utilization of the CPU. The utilization value is read periodically according to a *"sampling-rate"*. An *"up-threshold"* value is set to decide the average CPU utilization between two sample readings, at which the kernel needs to change the CPU frequency. The *On-demand* governor ramps up to the maximum frequency once the *"up-threshold"* is crossed. This ensures high responsiveness of the system. Later, it gradually reduces the frequency when detecting a decrease in the CPU load. The decrease of the frequency is proportional to a *"down_differential"* value. This governor is the default in most Android stock kernels, since it provides a good level of system performance. However, the fact that the *On-demand* jumps to the highest frequency on each load increase might consume an extra amount of energy which could be avoided.
- *Conservative*: The *Conservative* governor described in Algorithm 9, also sets the frequency dynamically based on the current utilization. However, it differs form the *On-demand*, that it does not jump to the maximum frequency on every increase. Instead, it gracefully increases the frequency according to a *"freq_step"* value. The *"freq_step"* defines the percentage step at which the CPU frequency will be smoothly increased or decreased by. For example, if it is set to 5%, the frequency will be increased at 5% chunks of the maximum frequency each time. Setting the *"freq_step"* to 100 will theoretically make it behave same as the *On-demand*. The *Conservative* governor is more battery-friendly since it avoids jumping to the maximum frequency unless when needed.

Algorithm 2: *Conservative* CPUFreq Governor

```
1 for every sampling_rate do
2     if cpu_load ≥ up_threshold then
3         target_freq + = (freq_step * max_load_freq ) / 100
4     else
5         if cpu_load ≺ down_threshold then
6             target_freq - = (freq_step * max_load_freq ) / 100
7         end
8     end
9 end
```

In the following section we will present the architecture and main components of mobile GPUs.

V. Mobile GPU Architecture

In mobiles, GPUs are located on the same chipset as rest of the processing cores, which is referred to as System-on-a-Chip (SoC) architecture [15]. Modern mobile GPUs, like CPUs, could be multi-core. Figure 2 displays the conceptual overview of the GPU rendering pipeline. An application running on the CPU would send requests to the GPU containing triangles that need to be rendered, along with additional rendering details. The Vertex Processing unit defines the positions of the vertices constructing the triangles. The Setup unit then takes these vertices and assembles a triangle of each three of them and computes the constant data over the triangle. Following that, each pixel in the triangle is processed to define its final visual appearance in the Pixel Processing stage. Finally, several buffer operations are performed, such as blending and resolving the different visibility levels of each triangle. The output may be written into memory if needed [5].

Although the functional design of the mobile GPUs is similar to the PC's, the limitations of mobile devices, such as size and battery, should be considered in the GPU design. Thus, many power reduction techniques are used on mobile GPUs to overcome these limitations. Examples of these techniques include clock gating, lowering memory bandwidth usage, energy-efficient arithmetic units, and the topic of our paper, frequency and voltage scaling.

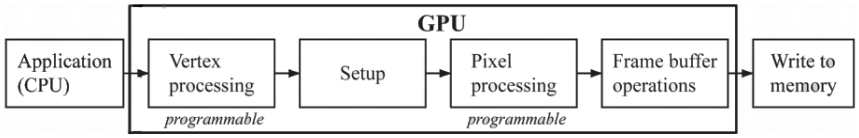


Fig. 2. Conceptual overview of GPU architecture [5]

VI. GPUFreq Scaling Governors

In this section, we will introduce the details of our GPUFreq governors and discuss the various elements and components of our design model.

A. Design and Architecture

Most silicon vendors of smartphones implement algorithms for dynamically scaling their devices' GPU. The algorithms vary depending on the underlying GPU and its device driver. However, the main goal of these algorithms is to provide a stable and consistent performance to maintain a general robust experience that satisfies majority of the customers. Therefore, such algorithms do not risk scaling into lower frequencies, and rather set the minimum at a higher frequency than needed. Moreover, for commercial purpose, these algorithms do not consider the vast variations between different usage patterns, and would only provide a single general interface that fits all. However, a class of users may be more concerned about the phone battery life, and are willing to endure a minimum loss in the graphical performance if it allows them to save battery. Another class of users that enjoy playing 3D-games and demand high graphical responsiveness of the system. Our proposed model of the GPUFreq scaling governors offer different types of users different alternatives to choose from. We prove that by giving users the control to tweak and customize their GPU scaling will result in a more granular and optimal energy saving, while maintaining the stability and robustness of the system.

1) Mobile GPU Device Driver: Drivers are the low-level software that communicates with a certain device providing an abstraction to the operating system for that hardware. We consider the device driver of the ARM Mali-400 MP GPU selected for the experiments in this study. Figure 3 shows the Mali-400 MP device driver stack. Mali-400 MP is one of the most powerful GPUs installed in smartphones. It support both 2D and 3D graphics, and its throughput can achieve 30M triangles/s, and up to 1.1G pixels/s at 275MHz [16]. There are two main parts of any GPU driver: the high level user space libraries, and the low level kernel space driver. The user space libraries provide APIs for 2D and 3D application developers. These APIs interact with the low-level kernel space driver which communicates directly with the GPU. The low level driver integrates as a layer in the given OS kernel. In this work we are more interested in this low-level driver, rather than the user space libraries which are considered out of the scope of this study.

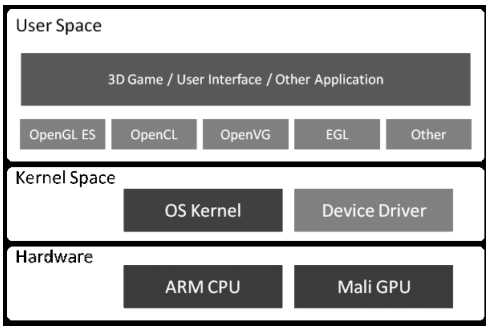


Fig. 3. Mali device driver stack [16]

The kernel GPU device driver provides low-level memory management, interrupt handling, and access to the GPU hardware. Moreover, the driver defines the set of accepted frequencies that a GPU can use, but can only accept a single frequency value at a given time. This leaves the task of dynamic runtime scaling to the framework or kernel. In addition, the GPU low-level device driver can be used to provide multiple run time information about the GPU that can be useful, such as the current GPU utilization or usage. Acquiring this value would enable the kernel to design a dynamic scaling algorithm according to the real-time utilization load.

2) Android Kernel Modification: For designing the GPUFreq scaling governors model we selected one of the most well-known and widely used mobile platforms, the Android OS. Android is a Linux-based operating system created by Google and Open Handset Alliance for touchscreen mobile devices. Android is an open source project released publicly under the Apache license; this allows manufactures, developers, and researchers to freely modify any part of Android and redistribute it.

The Android kernel relies on the Linux version 2.6. The main core system services of Android such as process management, network stack, and security are all Linux-based. In addition, the Linux kernel provides Android with its driver model. Linux acts as an abstraction layer between the software and the hardware components.

The Mali-400 MP GPU is installed in the Samsung Galaxy S2 GT-I9100, the phone we selected for our experiments (the full specifications of Samsung Galaxy S2 can be found on the official Samsung website). Samsung releases its specific Android kernel source code, that gave us access to the implementation of the Mali-400 GPU device driver of the kernel space. We based our implementation on the eighth update of the Samsung Galaxy S2 GT-I9100 ICS (Ice-Cream Sandwich)

TABLE I
SAMSUNG STOCK ALGORITHM FREQUENCIES AND VOLTAGES

Frequency (MHz)	Voltage (μV)
160	950000
267	1000000

TABLE II
GPUFREQ GOVERNORS FREQUENCIES AND VOLTAGES

Frequency (MHz)	Voltage (μV)
50	825000
62	825000
73	825000
80	825000
89	875000
100	875000
133	900000
160	950000
200	950000
267	970000

and kernel, which is the latest release for that particular device published by the Samsung Open Source Release Center at the time of the experiments [17].

To implement our model, we modified the source code of the Android Linux kernel to add a layer of GPUFreq scaling governors. Our implementation of GPUFreq governors layer makes it easy to plug in any additional custom governor, giving flexibility to our model design. The GPUFreq governors acquire the real time utilization value through an interface with the low-level GPU driver. Then the utilization is fed into the currently selected governor, and depending on its scaling algorithm and parameters, a corresponding frequency and voltage level will be selected. These two values are then sent back to the GPU driver to scale the GPU dynamically at run time. Figure 4 illustrates the design structure of the GPUFreq governors in Android. The Android kernel is built separately form the operating system using the *gcc* prebuilt tool-chain, and the resulting *zImage* can be packaged and flashed into the target device.

3) Frequencies and Voltages Selection: As mentioned previously, most smartphone silicon vendors prefer general GPU scaling algorithms that use a small number of scaling levels. Their aim is to provide the maximum graphical performance to claim boasting rights in the fiercely competitive smartphone spec wars. However, our work shows that allowing for more granular scaling with precise levels can provide energy-efficient scaling algorithms, with very minimal effect on the overall performance.

The original DVFS stock algorithm by Samsung for the Mali-400 MP in the orion-m400 platform scales between only two frequency values based on the real-time utilization of the GPU. The two values are 160 MHz, and the maximum frequency for that platform which is 267 MHz, as shown in Table I. This scaling between only two levels does not cover the wide variations of graphical needs of smartphone applications. Moreover, the lowest frequency used is 160 MHz, which we experimentally found to be much higher than what is required by many applications to run smoothly.

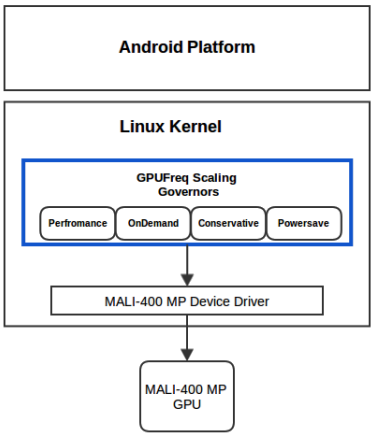


Fig. 4. Design structure of GPUFreq governors in Android

We have selected ten different granular levels of frequencies and their corresponding voltage to be used by our scaling governors. The frequencies were experimentally selected from the set of accepted frequencies of the Mali-400 GPU. Table II displays the new scaling levels. When the algorithm selects a frequency value, the corresponding voltage appropriate to that frequency is applied by the GPU regulator. As discussed earlier, higher frequencies require higher voltages, and vice versa.

B. Implementation

In our implementation of the GPUFreq governors, we aimed to maintain consistency with the original Linux CPUFreq governors in terms internal structure. The goal is to make it easy for those familiar with the CPUFreq governor model to understand and further tweak our implementation, if required. In addition, our model can easily be further expanded. If more optimal scaling algorithms are developed, they can easily be added as new governors options in having the same interface as the GPUFreq governors.

The governors implemented in this study are the four main scaling governors, *On-demand*, *Conservative*, *Performance* and *Powersave*. We further discuss the abstract details of our implementation of these four GPUFreq governors.

1) On-demand GPUFreq Governor: The *On-demand* governor provides high responsiveness since it scales up to the highest frequency whenever the load is increased. However, it also scales down gradually until reaching the minimum frequency that is considered sufficient by the current load. The GPUFreq *On-demand* resembles the structure of the CPUFreq governor shown in Algorithm 1. Four main tweakable parameters were added: *sampling_rate, up_threshold*, *sampling_down_factor,* and *freq_step*. The utilization value of the GPU is read every 1000 millisecond from the device driver. This value is then sent to the selected governor to decide on the next frequency and voltage level form the list in Table II.

Figures 5 and 6 show that the *On-demand* governor jumps to the highest frequency when needed and then gradually decreases the frequency to an acceptable level. We also illus-

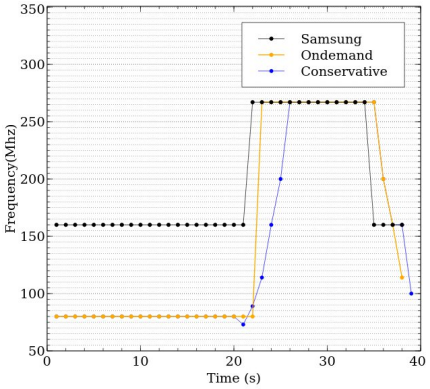


Fig. 5. The scaling algorithms in *AnTuTu* 2D Mode

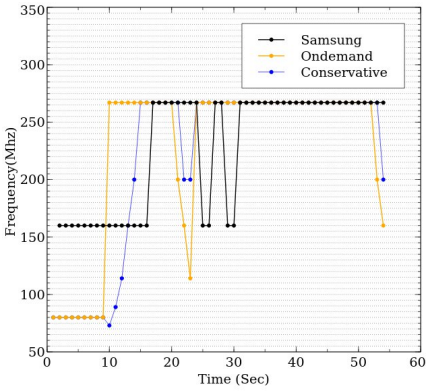


Fig. 6. The scaling algorithms in *AnTuTu* 3D Mode

trate the frequency scaling response of the Samsung algorithm for the same benchmark test, showing that it switches only between 160 MHz and 267 MHz.

2) Conservative GPUFreq Governor: The *Conservative* governor is recommended for battery-based devices, since it scales gradually and smoothly between frequency levels. The *Conservative* governor increases and decreases the frequency based on the *freq_step* tweakable value. Our implementation uses a *freq_step* of 5%, though it can adjusted. This implies that when the GPU utilization value reaches the *up_threshold* percentage, the frequency increases an additional 5% chunk of the maximum frequency. The GPU driver is responsible to find the valid frequency closest to the requested one. This ensures a gradual, energy-efficient scaling compared to the *On-demand* governor. Figures 5 and 6 show how the *Conservative* governor scales up and down between the frequency levels in gradual steps.

3) Performance GPUFreq Governor: To implement the *Performance* governor, we disabled the DVFS of the GPU and set the frequency statically to 267 MHz which is the highest for the used chipset. This governor is intended for users who require the maximum graphical performance from their devices in order to enjoy 3D graphics and games without any lag. In addition, this governor can be used for ranking and benchmarking purposes to stress test the performance level of the GPU.

4) Powersave GPUFreq Governor: The *Powersave* governor is the most power efficient, since it sets the frequency statically to the lowest value. However, we found that setting the GPU frequency to 50 MHz is not very practical since it slows down the rendering of some interface components, and it also fails to run some of the high resolution 3D games. Therefore, through experimental testing, we decided to choose a more reasonable frequency of 80 MHz for the *Powersave* governor. This governor is suitable for users who do not use their smartphones for displaying 3D graphics or playing 3D games. However, setting the frequency at 80 MHz does not effect any 2D graphic rending of regular mobile applications or GUI components.

VII. Experimental Setup

This section presents the testbed setup used in our experiments for evaluating the GPUFreq governors.

A. Benchmarks and User Applications

To evaluate the efficiency of our GPUFreq governors model, we used both graphical benchmark tests, and regular user applications. The graphical benchmarks can be considered as simulating users playing 2D or 3D games on the smartphone. For this we chose three of the most well-known graphical benchmark applications: *AnTuTu, Passmark,* and *FPS2D*. A total of 6 different benchmark tests were conducted, which cover both 2D and 3D modes. The tests were: *AnTuTu_2D, AnTuTu_3D, Passmark_Image_Filtering, Passmark_3D_Simple, Passmark_3D_Complex*, and finally *FPS2D*.

As for the regular user applications, we simulated the usage of some of the most well-known smartphone applications using the Android Monkey tool [18]. The Android Monkey is a command line tool that generates pseudo-random streams of user events simulating UI interaction. The Monkey tool is used via the ADB (Android Development Bridge). However, using the ADB requires connecting the device through USB to a workstation, which will result in the automatic charging of the battery, and that would invalidate our power measurements. As a work-around for the USB charging dilemma, we created multiple small batch files containing the Monkey script and used an internal android terminal emulator for executing the

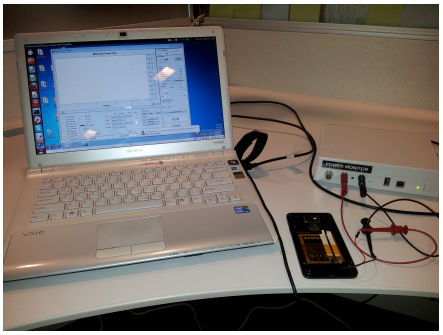


Fig. 7. Experimental setup: Monsoon Power Monitor

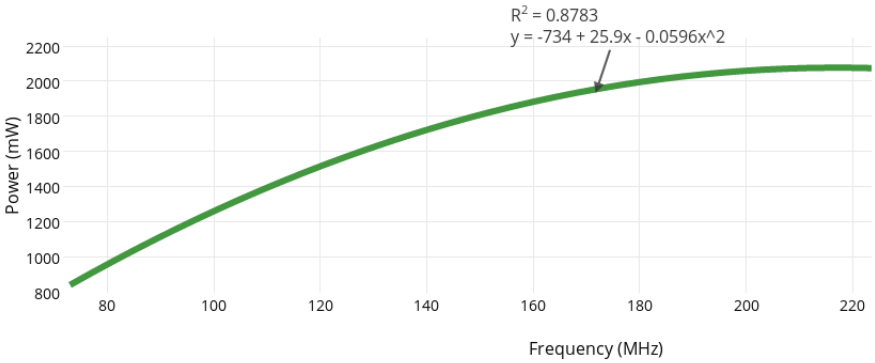


Fig. 8. Relation between the GPU frequency and the instantaneous power

batch files from within the device itself without the need for a USB connection.

We evaluated the following applications: Browser, Gmail, Facebook, Maps, and playing a HD video in full-screen mode.

B. Power Measurement

The Monsoon Power Monitor was used to measure the total consumed energy [19], with a sampling rate of 5kHz. The experimental setup of both software and hardware is displayed in Figure 7.

In order to eliminate any variations in power consumption of the testing environment, the following measures were taken:

- SIM card removed (GSM is off)
- Flight mode activated
- Bluetooth, Wifi, and 3G off.
- No running or sync applications in the background
- Lowest brightness of the display, with screen timeout disabled

C. GPU Monitoring Android App

For monitoring the different GPU and system parameters without the need for a USB connection, we developed an Android application with root privileges named the "*GPU-Logger*" which can be downloaded for free from our group website [20]. The *GPUlogger* is implemented as an Android service that works in the background, and reads system values at predefined time interval, then logs these values into a .csv file stored on the device for later retrieval. Some of these parameters are: *GPU frequency, GPU voltage, CPU frequency, CPU voltage, CPU usage, used memory, battery state, battery level, and battery temperature*.

VIII. Evaluation

This section presents the evaluation of the GPUFreq governors in terms of energy savings and performance using the previously mentioned experimental setup. However, first in order to fully understand the mechanism of the GPUFreq governors, Figure 8 displays a fitting function for varying the GPU frequency and its effect on the overall instantaneous power. The data was sampled while running the *AnTuTu_2D* using the *Conservative* governor. We observe that indeed when gradually increasing the frequency of the smartphone GPU the overall power of the entire phone increases accordingly.

The following subsections discuss the evaluation results for the GPUFreq governors in 3D Benchmarks, 2D Benchmarks, and User Applications. We evaluated each test measuring the total consumed energy in Joules, and the Frame-Per-Second (FPS) as our performance metric.

A. 3D Benchmarks

Figure 9 displays the energy and performance results for 3D benchmarks: *AnTuTu3D*, *PassMark_3D_Simple*, and *PassMark_3D_Complex*. In all three tests, the *Powersave* governor which sets the frequency statically to 80MHz is the most energy-efficient, saving 61.8%, 36.8%, and 33.2% of total energy compared with the original Samsung kernel. However, the *Powersave* fails to run 3D tests, giving more than 50% degradation in the FPS for all three tests. Therefore, the *Powersave* governor is not recommended for running 3D graphics or games.

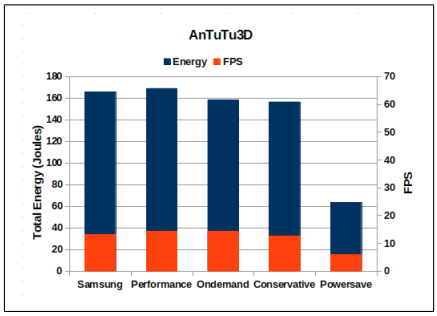
For more moderate scaling, the *Conservative* governor saves a considerable energy in 3D tests: 5.4%, 8.5%, and 6.9% in order, while keeping the FPS in an acceptable level. The decrease in FPS in the *Conservative* governor only shows a slight lag that dose not effect the overall experience for the user.

The *On-demand* governor saves around 4.4%, 3.9%, and 5.25% of energy in order, and performs even better than the original algorithm in terms of FPS. The energy saving is a result of the granular scaling the GPUFreq governors having 10 scaling levels instead of 2 which adjusts the power consumption very precisely.

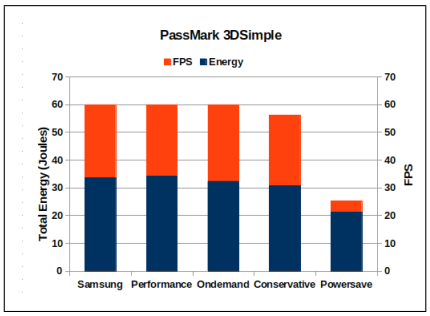
Finally, the *Performance* governor consumes around 2% extra energy compared with the stock algorithm. However, it offers a very smooth performance experience that gives the maximum potential of the GPU in 3D mode.

B. 2D Benchmarks

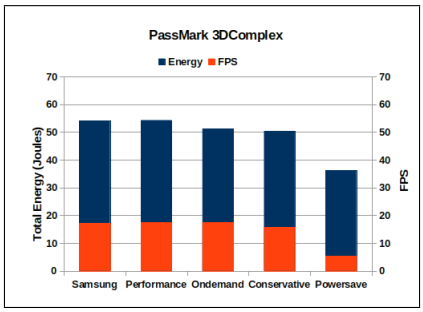
For our 2D benchmark tests, Figure 10 displays the total energy consumption and performance results of the four



(a) Antutu3D

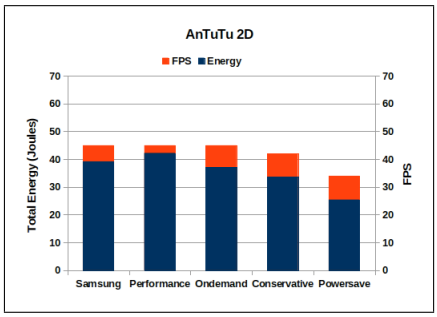


(b) PassMark 3D Simple

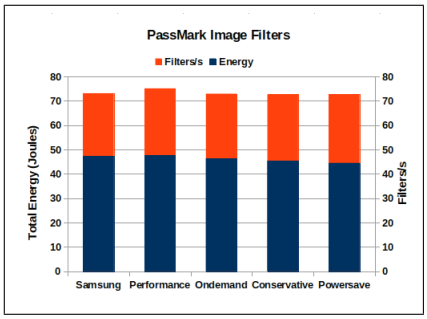


(c) PassMark 3D Complex

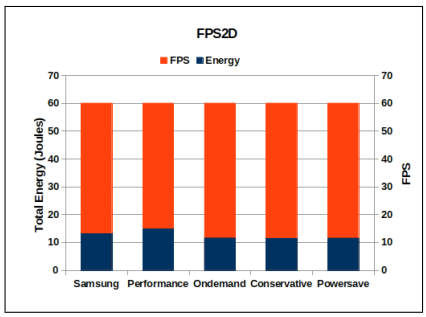
Fig. 9. Energy and performance comparison (3D Benchmarks)



(a) Antutu2D



(b) PassMark Image Filter



(c) FPS2D

Fig. 10. Energy and performance comparison (2D Benchmarks)

GPUFreq governors. The *Conservative* governor saves around 13.9% in *AnTuTu_2D*, as shown in Figure 10a, with less than 6% impact on the performance. This shows that the gradual scaling of the conservative governor in 2D can indeed save energy with minimal impact on the performance. The *On-demand* governor for the same test saves 5.5% with similar performance level as the Samsung stock algorithm. The *Powersave* governor saves 35.1% with around 24% drop in performance. The *Performance* governor consumes 7.7% additional energy for the same benchmark test without any noticeable increase in the FPS.

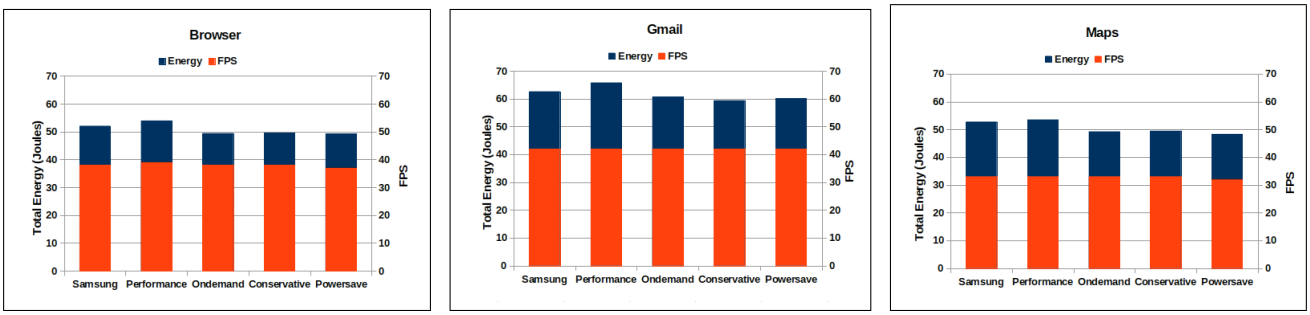
The *PassMark_Image_filters* evaluation results in Figure 10b show that the *Powersave* governor saves 6%, the *Conservative* 3.9%, and the *On-demand* 1.9% energy. The *Performance* governor consumes additional 1% of extra energy. In terms of performance, both the *Powersave* and *Conservative* governors showed less than 0.7% decrease in FPS, the *On-deamnd* displayed similar behavior to the stock algorithm, while the *Performance* governor showed an increase of 2.6% compared to the stock algorithm.

Finally, the *FPS2D* results in Figure 10c show that all four governors performed similarly in terms of FPS, with around 13.4% extra energy consumed by the *Performance* governor, and around 11% energy savings for the remaining three governors.

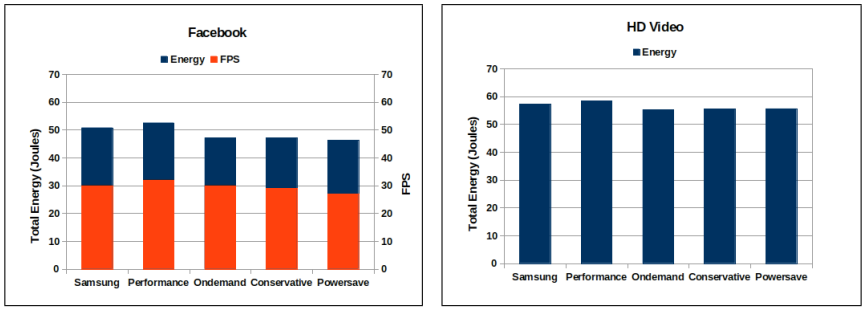
C. User Applications

We also evaluated the effectiveness of the GPUFreq governors when running regular well-known 2D user applications. The five applications selected for these tests are shown in Figure 11. We simulated the usage of each of these applications for a full minute, and then tested them against each one of the governors calculating the consumed energy and average FPS. Interestingly, our experiments showed that the GPU load required for rendering the GUI and images in these application did not reach the *up_threshold* defined for the two dynamically scaling algorithms. This means that both the *On-deamnd* and the *Conservative* governors ran these applications using their minimum scaling level. However, the applications were rendered very smoothly without any impact on the performance that can be noticed with the human eye. We conclude that the Samsung stock algorithm was utilizing the GPU at roughly twice the frequency and voltage required to run these common applications at a similar performance level.

The variation in the amount of energy saving between the five user applications depend on the amount and complexity of images that are requested to be rendered by the GPU. For the Browser application the *Powersave*, *Conservative*, and *On-demand* all saved around 5% of energy, while the *Performance* governor consumed around 3.6% of extra energy. The energy saving for the Gmail application for, the *On-*



(a) Browser (b) Gmail (c) Maps



(d) Facebook (e) HD Video

Fig. 11. Energy and performance comparison (User Applications)

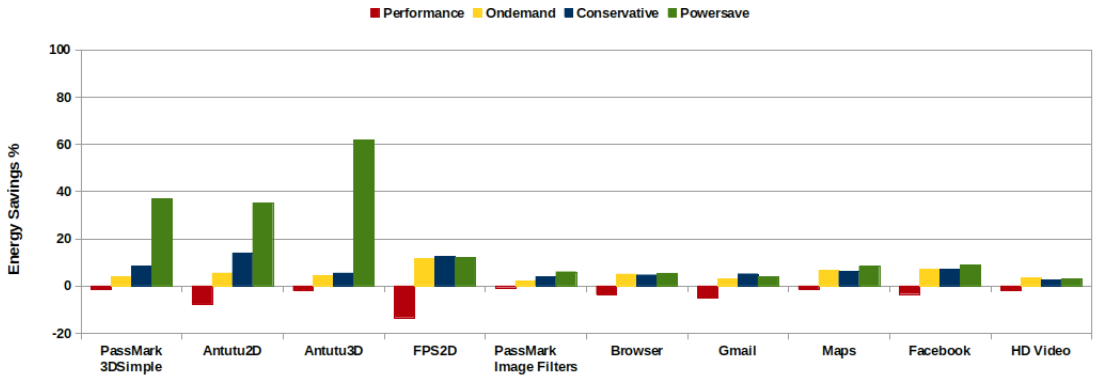


Fig. 12. Total energy saving (%) compared to the stock Samsung algorithm

demand saved 2.9%, the Conservative 5%, the *Powersave* 3.8%, while the *Performance* consumed 5% extra energy. For the Maps application, the *Performance* governor consumed only 1.4%, while the *On-demand* saved 6.6%, the Conservative 6.1% and finally the *Powersave* 8.4% of energy. For the Facebook application, the *On-demand* and *Conservative* both saved around 7% of energy, the *Powersave* saved 8.8%, and the *Performance* consumed 3.4% extra energy. Finally, we also evaluated the governors playing a HD video in full screen. Unfortunately we were unable to count the FPS while playing the HD Video since our measurement application views the video as a single frame. Nevertheless, we show the energy consumption results keeping in mind that there was absolutely no difference found running the video using the different governors in terms of performance. As for the energy savings the *On-demand*, *Conservative* and *Powersave* saved around 3% of energy, while the *Performance* consumed 1.9% more.

Figure 12 summarizes the total energy saving for all of our experimental tests. It is notable to mention that in our daily usage of the GPUFreq model, the battery lasted twice the time when using the default system. This practically proves that our model can save up to twice as much power compared to default scaling of the manufacturer.

IX. Conclusion

Smartphone GPUs are increasing in their computational and processing power. However, like other components in

a modern smartphone, they do not always need to run at their full capacity. With stock scheduling algorithms, users unforutnately pay a price in terms of degraded battery performance. We believe that users should be allowed the flexibility to scale the capabilities of different components of their phones according to their usage patterns. In this study we proposed the novel implementation of the GPUFreq scaling governors, similar to the Linux-based CPUFreq governors, to dynamically scale the frequency and voltage of smartphone GPUs. To the best of our knowledge our work is the first to present a quantitative analysis of the effectiveness of different DVFS algorithms tested on a real smartphone GPU, with the energy consumption accurately measured using an external power monitor device. Our results show that the energy-efficiency of the GPU can be enhanced up to 13% in 2D mode, and 9% in 3D mode with less than 7% effect on the performance using the *Conservative* governor. Moreover, using the *Powersave* governor would save up to 35% of energy in 2D mode, while maintaining an acceptable level of image rendering. And finally, the *On-demand* governor can save 11% in 2D mode and 5% in 3D mode with a higher graphical responsiveness. Our model was designed in a flexible and expandable fashion so that newer, and probably more efficient, governors algorithms can be added in the future.

REFERENCES

[1] Matt Wuebbling. Mobile graphics moving toward console level. http://blogs.nvidia.com/blog/2012/04/20/mobile-graphics-moving-toward-console-level/, April 20, 2012.

[2] Ahmad Rahmati, Angela Qian, and Lin Zhong. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, MobileHCI '07, New York, NY, USA, 2007. ACM.

[3] Hardware acceleration. http://developer.android.com/guide/topics/graphics/hardware-accel.html.

[4] Dominik Brodowski and Nico Golde. Linux cpufreq governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[5] Tomas Akenine-Moller and Jacob Strom. Graphics processing units for handhelds. *Proceedings of the IEEE*, 96(5):779–789, 2008.

[6] Jeff Pool, Anselmo Lastra, and Montek Singh. Energy-precision tradeoffs in mobile graphics processing units. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 60–67. IEEE, 2008.

[7] Victor Moya del Barrio, Carlos González, Jordi Roca, Agustín Fernández, and E Espasa. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 231–241. IEEE, 2006.

[8] Yan Gu, Samarjit Chakraborty, and Wei Tsang Ooi. Games are up for dvfs. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 598–603, 2006.

[9] Quake ii. http://www.idsoftware.com/gate.php.

[10] Bren Mochocki, Kanishka Lahiri, and Srihari Cadambi. Power analysis of mobile 3d graphics. In *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, volume 1, pages 1–6. IEEE, 2006.

[11] BVN Silpa, Gummidipudi Krishnaiah, and Preeti Ranjan Panda. Rank based dynamic voltage and frequency scaling fortiled graphics processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 3–12. ACM, 2010.

[12] Dirk Grunwald, Charles B. Morrey, III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[13] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228, 2010.

[14] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 89–102, New York, NY, USA, 2001. ACM.

[15] R. Zahir and P. Ewert. The medfield smartphone: Intel; architecture in a handheld form factor, 2013.

[16] Mali gpu device driver model. http://malideveloper.arm.com/develop-for-mali/drivers/mali-device-driver-model/.

[17] Samsung open source release center. http://opensource.samsung.com/.

[18] Android. Ui/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[19] Power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[20] KAUST Network Lab. Gpulogger android application package. http://www.shihada.com/packages/GPULogger.apk.