UNIVERSITY OF CALIFORNIA,
IRVINE

Android Application Level CPU DVFS Tuning

submitted in partial satisfaction of the requirements
for the degree of

MASTERS OF SCIENCE

in Computer Science

by

Sonny Wai-Git Lin

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Harry Xu
Professor Tony Givargis

2014

DEDICATION

To my family and grandparents for their enduring support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

ACKNOWLEDGMENTS

I'd like to acknowledge two people who motivated me to continue to explore the field of embedded systems. I would like to thank Professor Nikil Dutt, my advisor, for guiding and mentoring my studies in Android embedded systems. I gained a great respect and interesting to continue to understand and experiment with this field. Second I would also like to thank Jurngyu Park, for brainstorming and exploring ideas during this journey. It was truly an interesting experience to learn about the Android framework and Linux kernel.

I will also like to thank the committee for making time to read my thesis paper during their busy schedules.

Finally, I am truly grateful for the people who supported me during my graduate studies: Mom, Dad, Wilson Lin, David Dinh, Stephen Yang, and the Dutt Research Group (DRG). Thank you for everything!

ABSTRACT OF THE THESIS

Android Application Level CPU DVFS Tuning

By

Sonny Wai-Git Lin

Masters of Science in Computer Science

University of California, Irvine, 2014

Professor Nikil Dutt, Chair

Battery life and performance are two important aspects for smart phone devices. The Android platform runs on top of the Linux kernel. The Linux kernel allows Android users to tune or control the CPU settings via virtual governors and cpufreq in the application level. This thesis introduces an approach to tuning CPU DVFS Ondemand governor at the Android application level that allows better balance between the two aspects. This approach gathers information based on system sensors, application context, and CPU utilization to tune the Ondemand governor policy. Our approach allows users to tune their governor policies dynamically and without having to reinstall custom Android OS for their phones to achieve this balance. We compared the Ondemand and Interactive virtual governor settings to our approach for performance and power consumption. From our benchmarks, it is possible to achieve 8% to 17% power savings on idle state. For high single core CPU utilization, energy consumption improved for in a quad-core and dual-core system respectively by 7% to 13% without decrease in performance.

Chapter 1

Introduction

Smartphones are saturating the mobile device market with their increasing sales growth. Data from Gartner shows that global worldwide mobile phone sales have eclipsed feature phone sales for the first time in 2013 [1]. From this market share, the majority of the devices run on the Android platform. As of Q3 of 2013, the Android platform takes a 79% of the global smartphone market share [1]. With the increasing number of smartphones, there is a growing need to provide users the ability to effectively control their smartphone's power and performance.

The Android platform has the unique capability of providing users the configuration and information about their kernel's system status, since the Android framework runs on top of the Linux kernel layer, seen in Figure 1.1. Android users can take advantage of the kernel information normally seen in Linux desktops develop features that other mobile platforms are not able to utilize. With this architecture, numerous developers have created applications, such as Busybox in the Google Play Store, that are enabling users to have high customization and desktop capabilities [2].

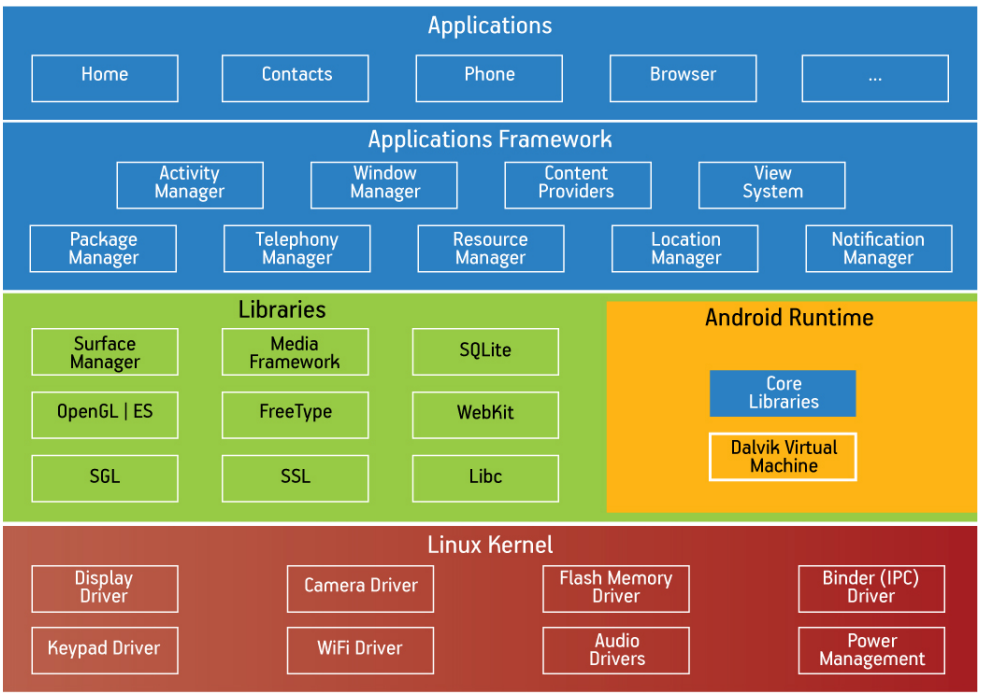


Figure 1.1: Android Framework [3]

Android applications allow users to adjust the balance between power and performance by controlling the smartphones with information from the kernel and application layer. At the application framework, programmers are able to allow users to control their devices' radios and CPU frequency. In this category, there is a subset of Android applications that focus on configuring the CPU frequency via Linux's `cpufreq` governors via the kernel layer [4]. From observations and surveys, these CPU applications statically configure the CPU frequency and CPU `cpufreq` governors based on the user configurations [5]. Although these applications allow users to set adjust both CPU frequencies and `cpufreq` governor, these static profiles do not address changes in utilization patterns. For instance, users leave their application running without interaction or running background tasks, can vary the performance and power savings. As a result, mobile devices do not have an effective way to dynamically tune the `cpufreq` governor based on CPU utilization. Situations where the applications enter high utilization or idle state, will result in performance loss or missed energy savings because the static frequency caps or incorrect selection of `cpufreq` governor.

In this thesis, we will present an approach with the Ondemand governor that allows users the ability to tune their governors based on utilization behavior. In Section 2, we will describe the surveyed approaches into smart phone power management and related works. In Section 3, we will present what we were tuning and how we implemented our approach. Section 4, we show our experiments on the Galaxy Nexus and Nexus 4 devices to determine the effect of the tuning the Ondemand governor. Finally, in Section 5, we will conclude our work regarding the results of our findings.

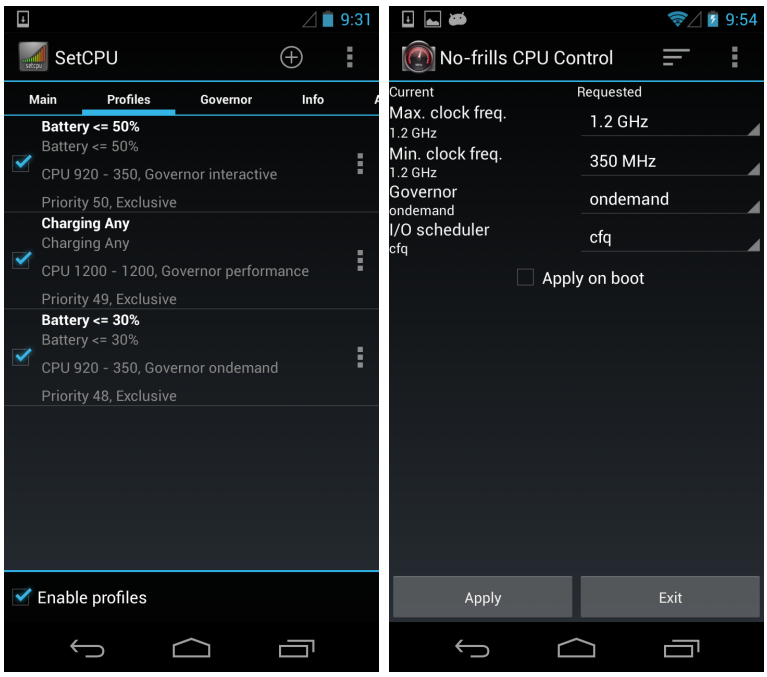
Chapter 2

Background

Our proposed approach is based on observations and survey papers found related to power management and savings on the Android platform. From our survey, the Android governors' tunable settings have many implications for the system performance and power savings. We will explore the effect of tuning the CPU DVFS governor as an alternative approach to achieve a balance between power and performance for mobile devices. This section covers the details of surveyed power management approaches in Android smartphones that motivated this research for tuning the CPU governor with application context.

Android users have different ways to approach this issue of power savings and performance. In the Android market, there are "battery saving" applications that focus on managing the Android system in the background. Such approach focuses on controlling GPS, Wi-Fi, and cellular radios via modeling of application behaviors or statically configured profiles [6] [7]. Another approach, like Carat by UC Berkeley, focuses on identifying what applications consume a significant amount of energy, so users can identify their application's characteristics to the community to identify the behaviors of certain apps [8].

Other approaches utilize administrative privileges that are available through the Linux kernel, to access and control the CPU frequency and `cpufreq` governor. Through administrative access, smartphone users can install and control system level configurations that are not available by default. Most notable of these applications that provide users the configuration between power and performance are SetCPU and No-frills CPU [9][10]. These applications provide basic adjustment of the CPU frequency and `cpufreq` governor, as seen in Figure 2.1, through static profiles or configurations. Based on static profiles and certain situations, these programs allow users to switch `cpufreq` governor and CPU frequency steps.



(a) SetCPU Profiles (b) No Frills Profiles

Figure 2.1: Static Configuration Profiles

In conjunction with applications controlling the CPU frequencies and `cpufreq` governor, Android users have been using custom kernels to improve their performance and energy usage. Android developer community provides custom Android operating systems, as a way to extend a smartphone's capabilities after the devices' intended end-of-life. CyanogenMod, a well-known group in this area, utilizes such approach static configurations on a global

context, similar to the No Frills approach in Figure 2.1 [12]. In the Android community there are numerous `cpufreq` governors that balance between the demands of performance and power with different approaches [13]. With applications, such as SetCpu, Android users can switch between governors and frequency steps with their own configurations. The other important factor for custom kernels and these type of application, is the increase frequency steps for the applications to select. Kernel developers introduce more frequency steps for their devices to increase performance or energy savings [11]. With these two approaches, current Android platform provides users a set of configurations for adjusting the CPU's performance and energy savings; however, the configurations can be cumbersome and do not reflect the system utilization due to running background tasks.

Related works in research also focus on creating custom `cpufreq` governors to apply certain theory or approach to CPU frequency selection. One research implemented a governor that switches frequency based on the frames per second (FPS) by tracking eglSwapBuffer calls in the Android systems [15]. They introduced a `cpufreq` governor that controls the CPU frequency only when it detects the application is a video game. Once their system detects a game application, they switch to their `cpufreq` governor and use a targeted a frame rate set by them to perform workload prediction to adjust the CPU cores' frequency. Chiou approaches the problem via monitoring the memory access rate of the system. The author(s) introduce the concept of critical speed for the execution time of a program to scale the CPU frequency based on the cache hit/miss rate ratio. With that information, the paper introduces a mathematical model called AD-DVFS that scale the frequency based on the cache hit/miss ratio [16]. Bezerra's research shows that `cpufreq` governors can consume more energy compared to statically set frequency for certain cases [17]. Their finding is that there are optimal frequencies for real applications, such as e-book readers and videos that provide better energy consumption compared to the Ondemand and Conservative governor [13]. Bezerra showed that setting optimal frequency steps for each application will reduce energy consumption; however, this finding does not mention about the potential performance

loss due to using static frequency. Prior research efforts acknowledge the idea that there are different factors to check, but they do not compare against the performance of the default governor such as Interactive or Ondemand.

Since most research surveyed focus on custom governors looking at specific feature sets, we realize that the default governor configurations will have a significant impact to the performance and energy savings. Zhang noted that mobile applications have varied power consumption due to the CPU's different idle state and workloads not fully utilizing multicore capabilities [18]. Also as noted earlier, the research by Bezerra, static frequency might provide better energy consumption. From our research, cpufreq governors have statically set parameters called tunables that affect the evaluation process of the `cpufreq` governor. Motivated by these researches and cpufreq Ondemand governor tunable parameters, we propose a new approach in dynamically tuning the cpufreq governor tunables to attain a balance between energy consumption and performance. Our implementation focuses on dynamically tuning the Ondemand governor tunables and frequency steps, based on system and application information provided by the Android framework and the Linux kernel.

Chapter 3

Approach

In the following section we will introduce the system we implemented in the Android application layer. We will then show the two default Android `cpufreq` governors and discuss their implementation.

3.1 System Design

From our observations and background research, we believe that a dynamically tuning `cpufreq` governor can allow potential power savings because it can adapt to the utilization characteristics of the system. This is important as the governors can be set to respond well to varied utilization, which results in higher energy consumption.On the other hand a conservative configuration results in slower CPU evaluations, at the cost of response to utilization spikes. Therefore, we introduce a tuning system from the Android application level that tunes the `cpufreq` frequency and governor based on the information collected in the system at the application level about the CPU utilization characteristics created by the running Ondemand governor. An overview of our system can be seen in Figure 3.1.

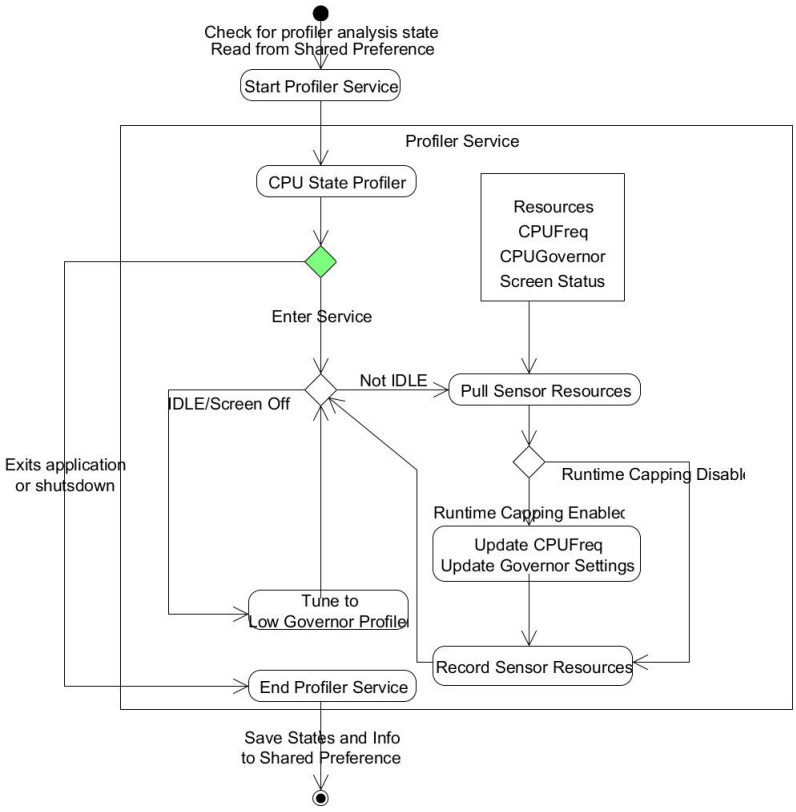


Figure 3.1: Profiler System

The motivation for the system to be at the application layer, or userspace layer, so that we can analyze the utilization patterns of the `cpufreq` governor for each application's activity state. Our tuning system can collect the following information about the Android system: CPU frequency, `cpufreq` governor, and screen status. We choose to analyze the CPU utilization, because we can identify the average usage of each core to adjust the governor's tunable parameters based on the retrieved CPU information. This allows the tuning system to have a holistic view of the running system, because it can identify when the scheduler switches between cores. This is possible on Linux system, because we can calculate the busy and idle utilization via `/proc/stat`, which shows information for each CPU core and the whole system. At each sampling interval on the system, we record the information into a circular buffer for each core to calculate a moving average that contains the CPU information for

the past samples certain amount of time. From this moving average, we are also able to calculate the variance of each CPU core and the entire system as a whole. The variance is important because the interactions on a smartphones are varied [19]. We categorized the system into three utilization states: Low, Mid, and High. We have the settings tuned to the following profiles for their respective categories, seen in Table 3.1. Since, this is at the application level, users are able to tune this at an application basis or configure the regions with different settings. For our purpose, Table 3.1 shows the baseline settings we are using for our tests and analysis to test our approach.

Ondemand	HIGH	MID	LOW
`sampling_rate` (µs)	90,000	150,000	300,000
`up_threshold`	75	80	90
`powersave_bias`	0	0	100 (400[1])
`max_frequency`	No Change	No Change	Decrease By 20%
`min_frequency`	Increase by 20%	No Change	No Change

Table 3.1: Ondemand Tuned Settings

Our approach categorizes the utilization into three regions because of the variations that could occur at each application context. Since the Android system allows running background tasks, this approach can compensate for overhead caused by active background tasks. The approach focuses on five important configurable parameters that are available by the `cpufreq` interface and the Ondemand governor. From our research about the Ondemand governor, we focus the three tunable parameters: `sampling_rate`, `up_threshold`, and `powersave_bias`. These three tunables primarily affect the governor's evaluation process to control the CPU frequency during each sampling of the CPU core. We will explain how we decide on the tunable values in section 3.2.2. For low and high regions, we control either the max or min frequency steps that the `cpufreq` governor can scale to via `scaling_min_frequency` and `scaling_max_frequency`. The reasoning for this is to avoid

degradation in performance or power consumption due to characteristics of the Ondemand governor in evaluating the CPU utilization and when the scheduler switches jobs between CPU cores.

During runtime, the system evaluates and retains the global and application historical utilization average, utilization variance, and frequency average to determine the region to select. From the calculations, the system adjusts the tunable governor and `cpufreq's` max/min frequency with the preset user defined settings. The pseudo logic can be seen in Algorithm 1.

Algorithm 1 Tuner logic

while *ServiceRuns* **do**
  for each core **do**
    if core utilization > 0 **then**
      Based on frequency average, CPU utilization average, utilization variance find category region and max/min frequency steps
    end if
    Update core `cpufreq` max and min frequency
  end for
  Pick the highest state determined from evaluation of each core
  if Utlization reflects the region characteristics and variance is within a predefined threshold **then**
    Select the region based on the utilization for the CPU Ondemand governor
    Based on application's average utilization set `sampling_rate`, `up_threshold`, and `powersave_bias`.
  end if
end while

The algorithm runs on the Android Java layer as background service. We do this as accessing the information that are provided by the Android SDK will allow retrieval of information regarding the running application stack and screen status. Initially, we did consider tracking input by adding a notification system service in the Android framework, but the overhead was varied and insignificant. This occurs when users touch and hold on the screen, which results in a high number of input events. We were able to reduce the CPU utilization overhead of our tuner by reducing the number of instantiated objects in the Java layer during the run time of the tuning mechanism. The overhead overall was around 2%. The duration of the decision process takes about 3 to 10 milliseconds to complete. Figure 3.2 shows a high level overview of how the tuner interacts with the CPU cores Ondemand `cpufreq` governor. The implementation is available via a Bitbucket link in Appendix A.

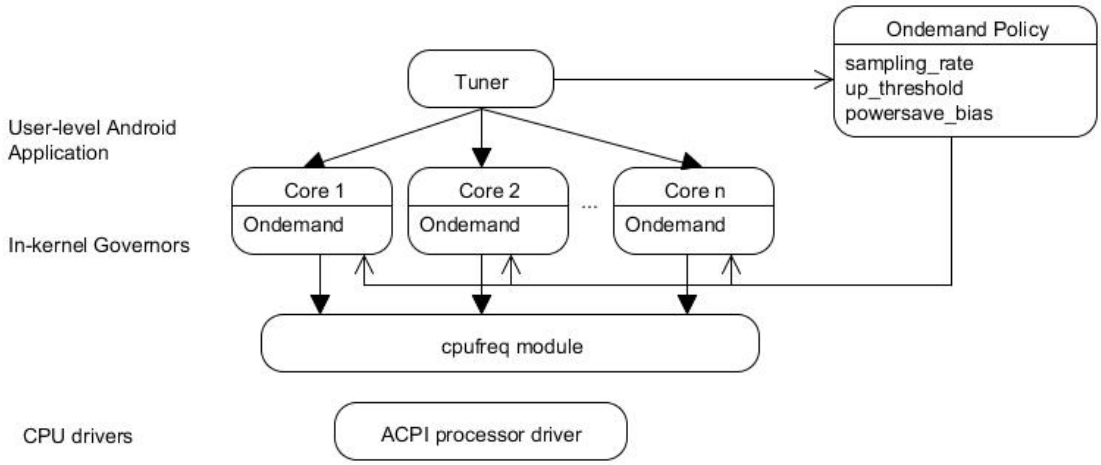


Figure 3.2: System Infrastructure

3.2 CPU Governor

In the Linux kernel, programmers use the `cpufreq` interface to create governors that can dynamically scale the CPU frequency based on a theory or idea[4]. We will explain briefly about the two governors, Interactive and Ondemand, and characteristics to provide some

ideas of the capabilities of the governors, while showing the difference in tunables on the Galaxy Nexus and Nexus 4 [20][21].

3.2.1 Interactive Governor

Interactive	Galaxy Nexus	Nexus 4
`above_highspeed_delay`	100000	20000
`boost`	None	None
`go_highspeed_load`	50	85
`hispeed_freq`	70000	1512000
`input_boost`	0	0
`min_sample_time`	60000	80000
`timer_rate`	20000	20000

Table 3.2: Interactive Governor settings.

The Interactive governor, introduced by Google in 2010, is a polling governor that evaluates the CPU workload at a sample interval and when the CPU wakes from idle [22]. The purpose of this governor was to improve responsiveness to latency-sensitive workloads. This is evaluation is done via a statically defined table called `target_loads` that defines a certain frequency for a range of CPU load [4]. As a result, research has shown that the Interactive governor is more responsive in latency workloads compared to the Ondemand governor [23]. Although, `target_loads` parameter is not available for application programmers and users, there are other tunable parameters that affect the balance of performance and energy consumption. The default static configurations for the Nexus 4 and Galaxy Nexus are available in Table 3.2.

From the table 3.2, the configurations show that the manufactures have different settings for the `cpufreq` governors. In the Galaxy Nexus, the `hispeed_freq` frequency is set to be 700 MHz while the Nexus 4 is the maximum frequency of 1.5 GHz. The governor scales the frequency whenever the load exceeds the value set by `go_highspeed_load`. However, the `hispeed_freq` is also used by two tunable parameters, `boostpulse` and `input_boost`. Both devices use `boostpulse` or `input_boost` to scale the frequency immediately when an input event from the touch screen is encountered. As seen in the Nexus 4, this scales to the maximum frequency step. The governor scales to the target frequency to reduce time for starting or interacting with applications, based on the assumption that touch inputs preemptively starts intensive work loads. In summary, the Interactive governor provides the best performance because of how proactively it scales the frequency and scale based on input events. However, this means excessive energy usage occurs due to preemptively scaling the frequency on every input event.

3.2.2 Ondemand Governor

Ondemand	Galaxy Nexus	Nexus 4
`ignore_nice_load`	0	0
`io_is_busy`	0	1
`powersave_bias`	0	0
`sampling_down_factor`	60000	80000
`sampling_rate` (µs)	300000	50000
`sampling_rate_min` (µs)	30000	10000
`up_threshold`	95	90
`down_differential`	N/A	3

Table 3.3: Ondemand Governor settings.

The Ondemand governor is a `cpufreq` governor introduced to Linux operating system by Intel [14]. This governor is widely adopted and functions less aggressively compared to the Interactive governor. The governor only calculates the utilization of the target core at a sampling interval set by the `sampling_rate`. The Ondemand governor scales to the maximum frequency, once the CPU utilization exceeds the `up_threshold` value. Downscaling to a lower frequency is determined at the next sample interval with respect to CPU utilization load.

It is also important to note that the Ondemand governor has received some tunable setting changes from Linux kernel 3.0 to 3.4. Intel introduced the idea of changing the CPU frequency based on sampling window improve the frequency downscaling decision [24]. Although, there is a subtle change in calculating the frequency average, the kernel still uses the same algorithm to determine the downscale frequency in both Ondemand governor in the Galaxy Nexus and Nexus 4, seen in Figure 3.3. With this algorithm, the users are still able to tune a set of tunables that are available from the two devices.

```
cur_load = 100 * (wall_time - idle_time) / wall_time; // Busy time
max_load_freq = cur_load * freq_avg;
freq_next = max_load_freq /(up_threshold - down_differential);
```

Figure 3.3: Ondemand Frequency Downscaling Method [25][26]

Comparing the two devices' `cpufreq` Ondemand governor tunables we see a significant difference in the `sample_rate` and `up_threshold` values. In context, the Linux kernel clock tick is approximately 10 milliseconds [27]. The Nexus 4's Ondemand governor evaluates the CPU core every 50 milliseconds, while Galaxy Nexus's evaluates the CPU core every 300 milliseconds. From this difference, we see that the Ondemand governor's sampling rate affects the load on the system because of how often it evaluates the CPU load.

From section 3.1, our approach focuses on tuning the `sampling_rate`, `up_threshold`, and `powersave_bias` into three categories: Low, Mid, and High. Using the static configurations

for each region, we allow variability in the Ondemand governor's evaluation process.In high utilization region, the higher `sampling_rate` and lower `up_threshold` values are to improve response times to utilization spikes For mid level category, we decided to provide the system a medium setting of the two devices, while lowering the `up_threshold` to increase responsiveness of the system if the utilization exceeds the application's characteristics. In the low categorization, we increased the `sampling_rate` and `up_threshold` to decrease the chances of scaling to the max frequency. In addition, we also manipulate the `powersave_bias` to avoid scaling to the max frequency, until the utilization characteristics exceeds the application's utilization characteristics.

Chapter 4

Experiments

4.1 Experimental Setup

Our experiments setup is tested on two mobile devices, the Galaxy Nexus and Nexus 4 [20][21]. We use a Monsoon Power Monitor to measure the energy consumption on the two devices [28]. We created a work flow, seen in Figure 4.1, to allow repetitive tests on the Android devices.The PC starts each tests via an AutoIt script [29]. At the start of the test, it starts a batch file that passes the tests parameters into MonkeyRunner sets up the Android application via a python file benchmark-start.py. We record the performance for our benchmarks utilize a combination of Android SDK MonkeyRunner and RepetiTouch Pro to start or to simulate rapid inputs for each test [30][31]. Once the startup of the testing application completes, the AutoIt3 script disconnects the USB via an autopass control from the Monsoon Power Monitor. After the duration of the test ends the script stops and reconnects the power monitor to the PC. The final step, the script calls MonkeyRunner to take screenshot of the results, exit the program, and to record the Monsoon Power Monitor information onto the PC.

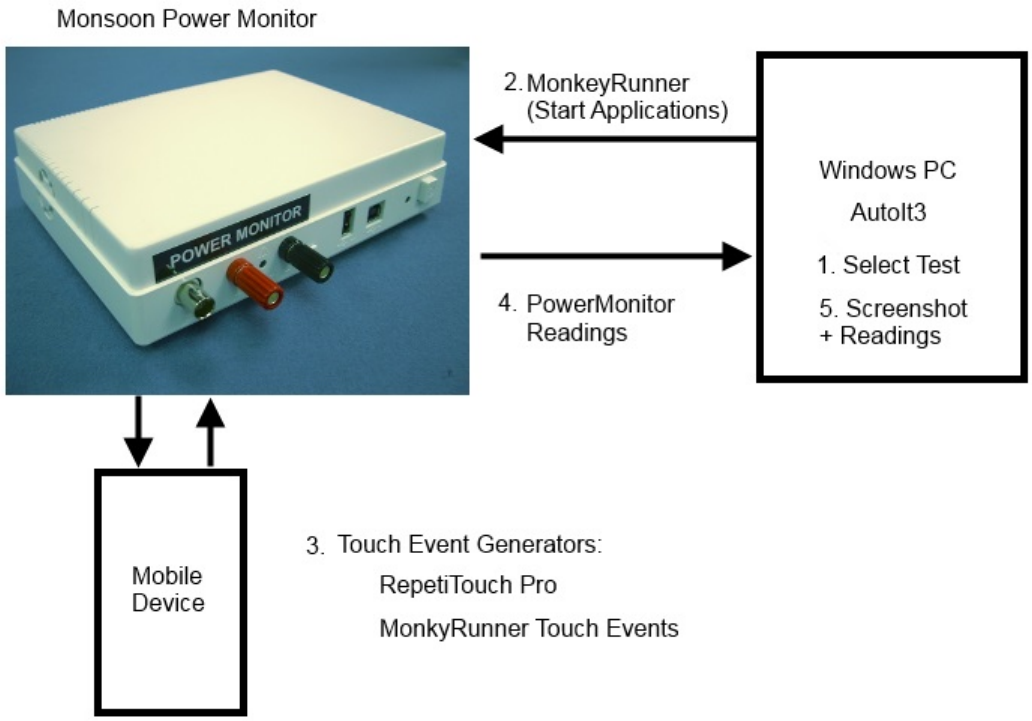


Figure 4.1: Testing Setup

The two devices both run on Android version 4.2.2. However, the two devices have fundamental differences in their specs and governor configuration. The hardware differences are shown in Table 4.1.

	Nexus 4	Galaxy Nexus
CPU	Quad-core 1.5 GHz Krait	Dual-core 1.2 GHz Cortex-A9
GPU	Adreno 320	PowerVR SGX540
RAM	2 GB	1 GB
Linux Kernel	3.4	3.0
Voltage	3.7 V	3.8 V

Table 4.1: Nexus 4 and Galaxy Nexus Hardware Specification [20][21]

These differences seen in Table 4.1 primarily affect the results for high and mid utilization tests. The Nexus 4 has a powerful CPU and GPU chipset, which allows rendering and execution time to be short. On the other hand the Galaxy Nexus's hardware is less power, which could result in better energy consumption because it has less CPU and GPU cores. The most important aspect for our test is the software side of the two devices. One is the Linux kernel version, the Galaxy Nexus and LG Nexus 4 runs on the Linux kernel 3.0 and 3.4 respectively. As mentioned earlier in section 3.2.2, the kernel change also affects the default Ondemand governor's down scaling methodology, such that in kernel 3.4 the next frequency is evaluated with an average frequency window [24]. However, the most significant difference, is the governor's tunable parameters. The difference in `up_threshold` and `sampling_rate` values affects the performance and energy consumption, which we will see in the results of our experiments. Finally, the are frequency steps available in the Nexus 4 compared to the Galaxy Nexus, seen in Table 4.2. The more frequency steps available, there is more opportunity to downscale to a frequency step that can work with the utilization load.

Nexus 4 (Hz)	Galaxy Nexus (Hz)
38400 486000 594000 702000 810000 918000 1026000 1134000 1242000 1350000 1458000 1512000	350000 700000 920000 1200000

Table 4.2: Nexus 4 and Galaxy Nexus Frequency Steps

4.2 Results

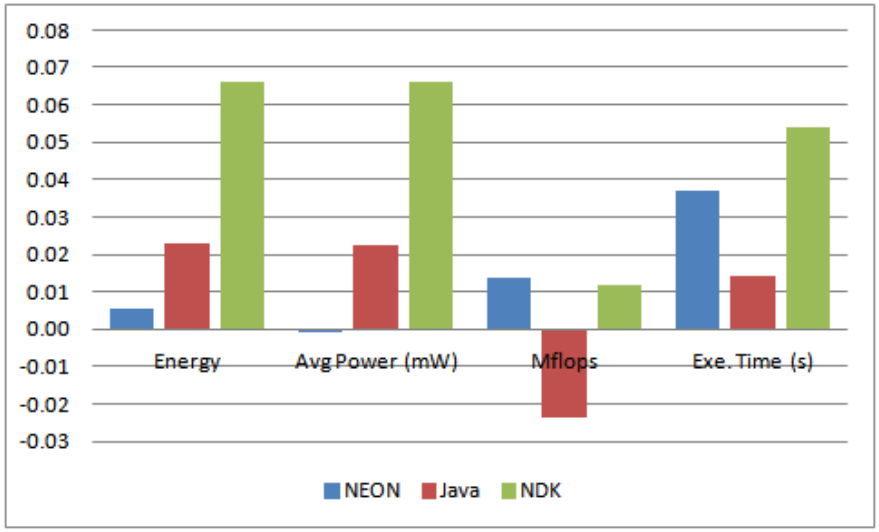
We tested our approach by testing high, mid, and idle CPU loads. This approach allows us to see if the tuning system has an effect compared to a statically tuned system in various CPU utilization scenarios. For instance, the idle test allows us to test if using `power_bias` will help reduce power consumption when the screen is off or running low utilization tasks.

We will show the difference with the tuned Ondemand governor as the baseline. The actual measured values from the Monsoon Power Monitor are in Appendix B.

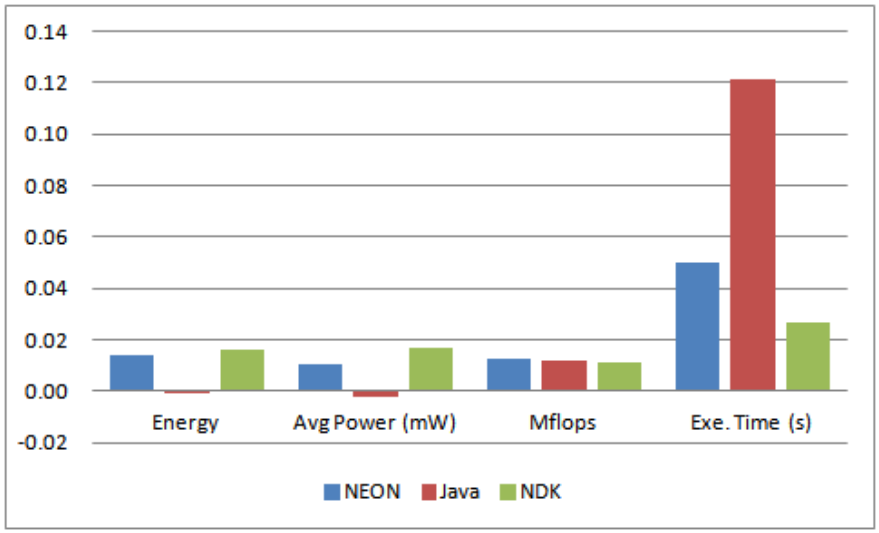
4.2.1 High Utilization

For high utilization tests we choose to use Linpack for its high single core utilization. Linpack is a benchmark designed to perform linear algebra computations that solves linear equations and linear least-squares problems [32]. We utilize the open source Android Linpack benchmarks created by Roy Longbottom [33]. These benchmarks are available in NDK, Neon, and Java for Android [34][35][33]. These set of benchmark are a set of linear algebra matrix computations that has an execution time between 5-10 seconds. Figures 4.2 and 4.3 provide the energy and performance results that were seen after ten iterations with the Linpack tests.

Nexus 4



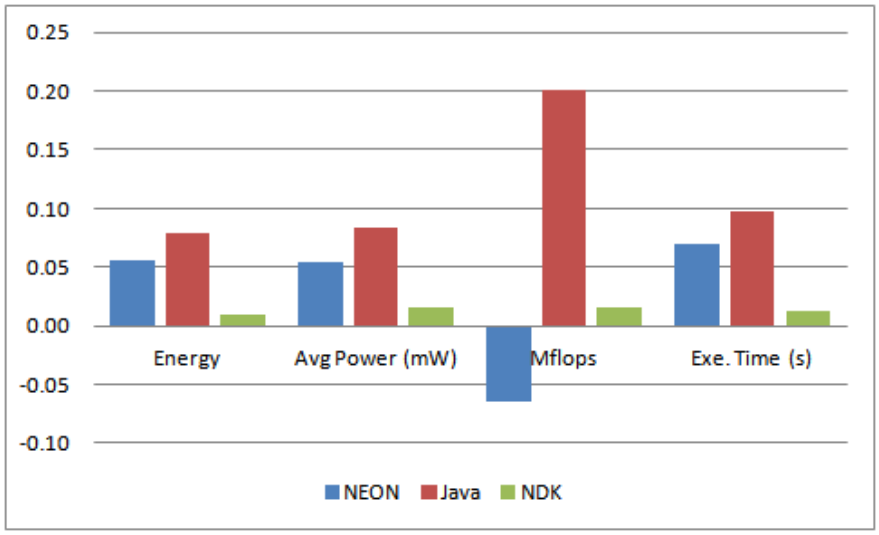
(a) Nexus 4 Ondemand/Tuner Linpack



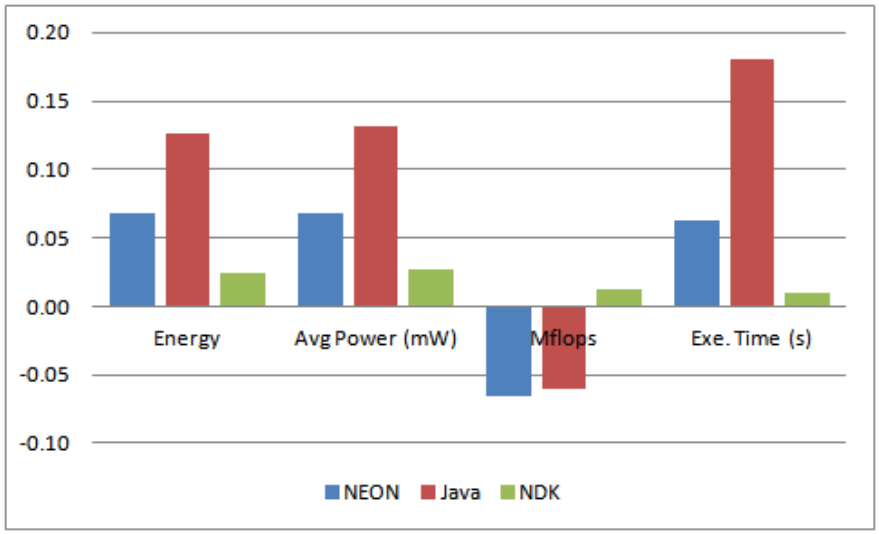
(b) Nexus 4 Interactive/Tuner Linpack

Figure 4.2: Linpack Nexus 4

Galaxy Nexus



(a) Galaxy Nexus Ondemand/Tuner Linpack



(b) Galaxy Nexus Interactive/Tuner Linpack

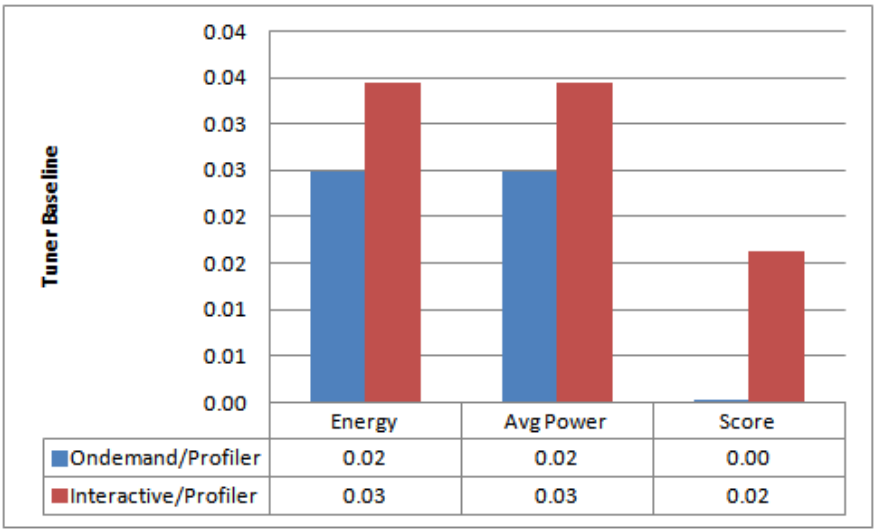
Figure 4.3: Linpack

From these three tests on the two devices, we see that there are power energy savings or performance improvements for different parts of the programming layers. We observed from traceview, that the improvement is due to the cost of switching between CPU cores [36]. During some tests, we noticed that the running task have the tendency to switch to another CPU core during its execution. As a result, the core takes time to ramp up back to its

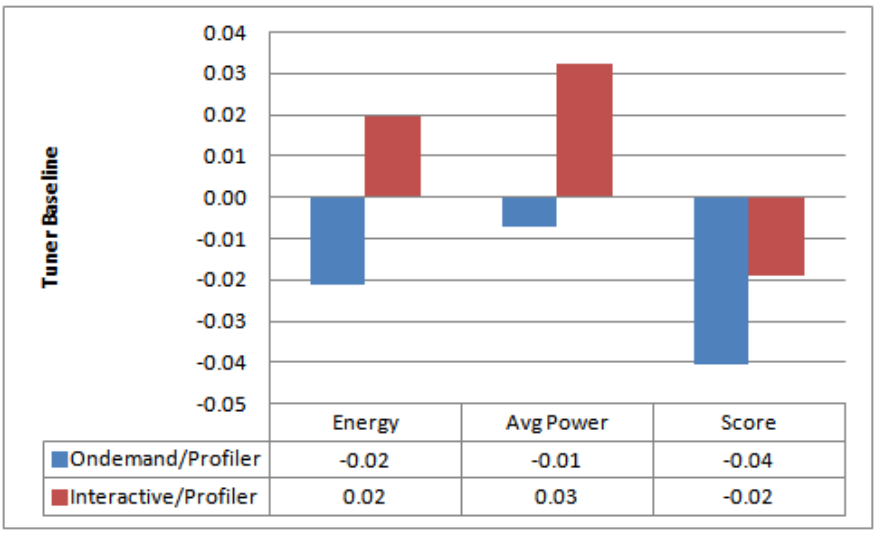
effective frequency. We can see on the Nexus 4 the improvement is smaller comparison, primarily due to the higher `sampling_rate`, which will be able to address such core switch at a shorter period. Since we preemptively shift the minimum frequency to be slightly higher than the lowest frequency step for the active CPU core; we were able improve the execution time and reduce the power consumption for the Linpack cases without impact to the overall performance. In some cases, such as in Galaxy Nexus's Java Linpack, we were able to improve the energy efficiency and performance of the test.

4.2.2 Mid Utilization

Antutu



(a) Nexus 4 Antutu



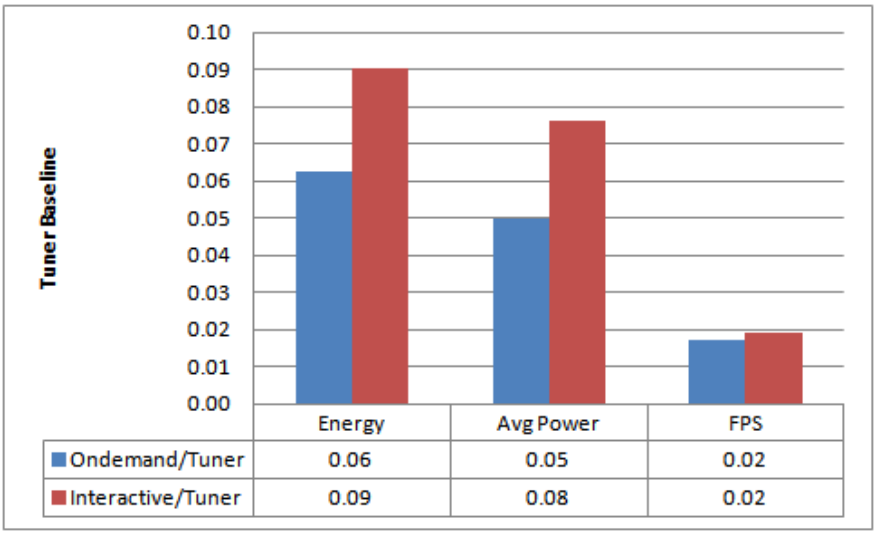
(b) Galaxy Nexus Antutu

Figure 4.4: Antutu Data

The Antutu benchmark is a common benchmark used in Android platforms to determine the capabilities of a smartphone Antutu. This test composes of CPU, memory, I/O, and graphics tests. We choose this test as a high to mid utilization benchmark test. The test

is to see how a real benchmark tests performance and energy consumption be affected if we dynamically change the `sampling_rate` and `up_threshold` based on the load the tests that were performed by during the execution. Our testing application runs ten consecutive tests on both devices with duration of 175 seconds on the Nexus 4 and 195 seconds on the Galaxy Nexus. The test time duration is result of the GPUs on both devices which affects the Antutus OpenGL tests. From the results, we see there is 2% to 3% improvement in energy consumption for the Nexus 4, with minimal performance difference. We can see that the high `sampling_rate` of the default settings allow the governors to adapt to the workloads effectively, which minimized potential performance gains. However, on the Galaxy Nexus we see that this is a different case. Our tuner gained 4% and 2% improvement in the Antutu score, compared to Ondemand and Interactive governors' settings; however, the energy consumption appears to be in between the two governors. The increase and decrease from the base `sampling_rate` with our tuner on both devices results in the difference in energy and performance seen in our test results.

Angry Birds



(a) Nexus 4 Angry Birds

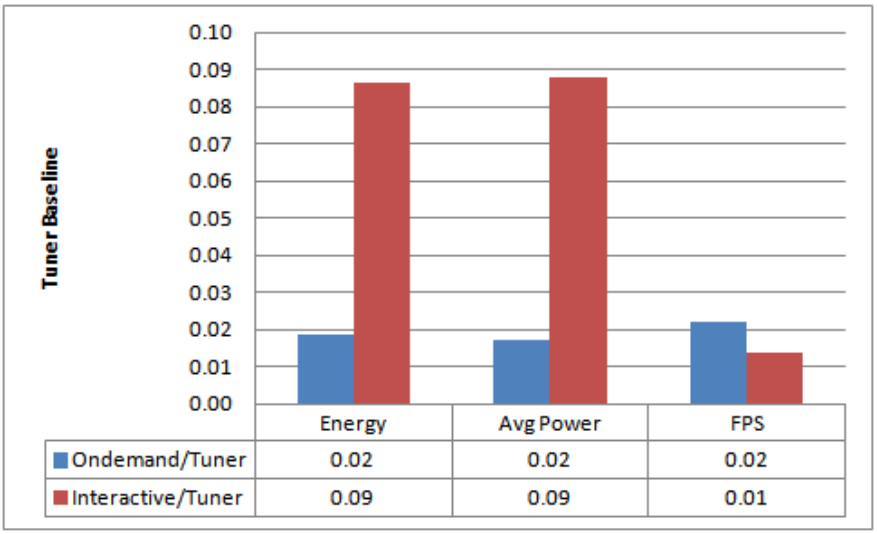


Figure 4.5: Galaxy Nexus Angry Birds

Figure 4.6: Angry Birds

Angry Birds is a low-mid CPU utilization test. Using RepetiTouch, we repeatedly load the game, play the first level, and exit the game [31]. For our performance metric we used FPS Meter, which is an Android application that provides a one minute frames per second (FPS) average [38].

The purpose is to measure the effect of the sampling rate to its light load and sudden increases in CPU utilization. The primary motivation for this test is to compare how much energy can be saved when the CPU utilization resides primarily in betweens the Low and Mid utilization regions with our tuner switching the `sampling_rate`. This test shows an important aspect in the difference in the `cpufreq` governor tunable settings for both devices.

For the Interactive governor, on both devices, the tuner gains a 9% energy savings with small loss in FPS comparing to the tuner system. The Interactive governor's hair trigger call `input_boost` or `boost` that is the primary reason for the high energy consumption; this trigger is called whenever an input is detected on the touch screen. This parameter immediately scales the frequency to the frequency defined by `hispeed_freq`. Since this is a heavily touched-based application, the Interactive governor is prone to scale the frequency set in `hispeed_freq`. This result in higher consumption with a small increase in FPS compared to our tuner in both devices.

For the Ondemand governor on the two devices, the energy consumption provides a different perspective of the effect of the tuner. On the Nexus 4 the tuner was able to improve energy consumption by 6%, while the improvement in the Galaxy Nexus was around 2%. Since the `sampling_rate` on the Nexus 4 is six times faster than the Galaxy Nexus `sampling_rate`, it is easier to scale to the maximum frequency. Reducing the `sampling_rate` at lower utilizations reduced the volatility of scaling frequency seen in the Nexus 4, resulting in better energy consumption. However, on the Galaxy Nexus our improvement can be attributed to the increasing the `sampling_rate` because of the transitions to a higher utilization region during the tests, similar to what we saw in the Linpack tests.

MiBench

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	967.11	1,076.09	1,013.03	-4.53%	6.23%
Avg. Power (mW)	1,201.28	1,322.53	1,277.54	-5.97%	3.52%
Total Test (s)	4.96	3.67	4.88	1.64%	-24.78%
Num. Tests	91.80	119.00	93.78	-2.11%	26.90%
Avg Test Time (s)	0.05	0.03	0.05	5.22%	-40.13%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	870.24	976.45	978.79	-11.09%	-0.24%
Avg. Power (mW)	1,233.24	1,266.46	1,246.67	-1.08%	1.59%
Expected Life (hr.)	6.47	6.30	6.40	1.09%	-1.50%
Total Test Time (s)	4.46	3.22	3.55	25.60%	-9.51%
Num. Tests	101.20	124.80	114.60	-11.69%	8.90%
Avg Test Time (s)	0.04	0.03	0.03	42.23%	-16.88%

Table 4.3: MiBench FFT

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,003.22	1,073.64	1,008.04	-0.48%	6.51%
Avg. Power (mW)	1,263.76	1,354.28	1,283.62	-1.55%	5.50%
Expected Life (hr.)	6.18	5.73	6.05	2.12%	-5.22%
Total Test Time (s)	5.21	4.66	5.32	-2.18%	-12.40%
Num. Tests	32.60	35.00	32.40	0.62%	8.02%
Avg Test Time (s)	0.16	0.13	0.16	-2.92%	-19.54%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	890.43	1,006.59	1,035.81	-14.04%	-2.82%
Avg. Power (mW)	1,312.68	1,279.34	1,321.58	-0.67%	-3.20%
Expected Life (hr.)	6.07	6.23	6.03	0.70%	3.32%
Total Test Time (s)	5.39	5.25	5.07	6.23%	3.55%
Num. Tests	30.80	32.00	32.20	-4.35%	-0.62%
Avg Test Time (s)	0.17	0.16	0.16	11.06%	4.17%

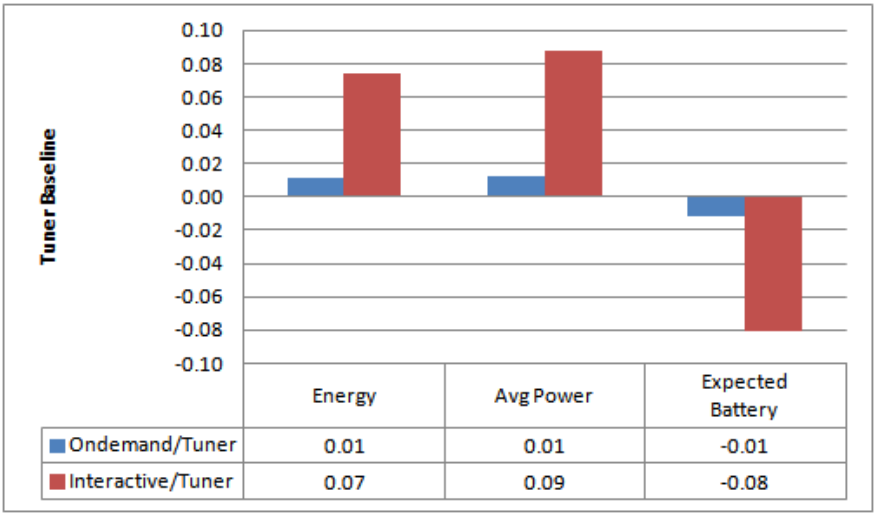
Table 4.4: MiBench Basicmath

In order to simulate varied interval utilization, we created a micro-benchmark utilizing a subset of the MiBench embedded tests suite [39]. We used the fft_small and basicmath_small tests, because their execution times were small. The fft_small test completes around 30 to 50 milliseconds, while basicmath takes around 150 to 180 milliseconds. When each test is completed the running thread sleeps for 50 and 150 milliseconds respectively to simulate idle times and test the effect it has on the governor policy. We run the two tests ten times for duration of ten seconds, to simulate a constant repetitive low and medium workload. Our motivation for this is to evaluate the effect of tuning both the `sampling_rate` and `up_threshold` at different intervals, because the Ondemand governor will evaluate the load

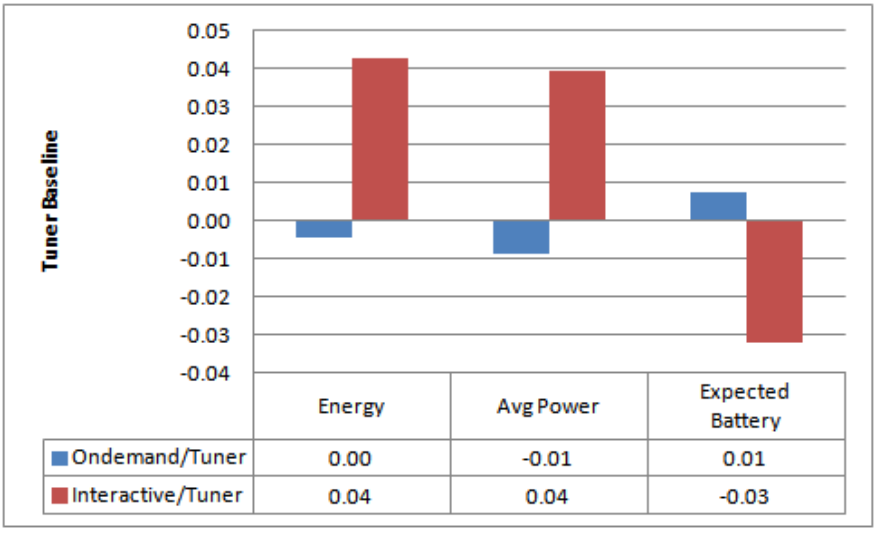
differently. Since this is a content repetitive test, the results are not as uniform compared to the Antutu and Angry Birds test because it does not reflect the interactions seen in real applications; however, the results reflect the approachs change in `sampling_rate` regarding the performance and energy consumption. From the Galaxy Nexus we see that the test showed the tuner to be in between the Interactive and Ondemand governor's performance and energy consumption. On the Nexus 4 the tuner consistently resulted in the highest energy consumption, but with the best performance. Since two devices have different sample rate the modified MiBench benchmark show that the `sampling_rate` of the governor affects the energy consumption and performance during these repetitive tests. With the change in `sampling_rate`, the benchmark shows that the number of test do vary; however, with the tuner's approach our average test times are within 20 milliseconds comparing to the average test times of the static settings of Ondemand governor. From these MiBench results, we see that by tuning the `sampling_rate` and `up_threshold` can achieve a balance between power and performance.

4.2.3 Low Utilization

Audio



(a) Pink Noise Nexus 4

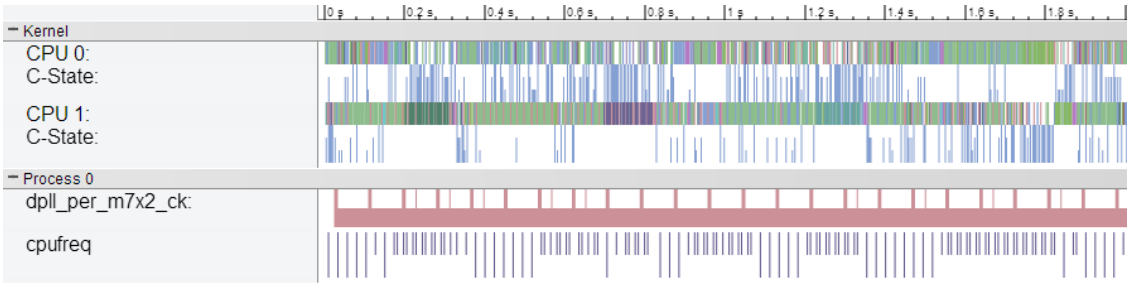


(b) Pink Noise Galaxy Nexus 4

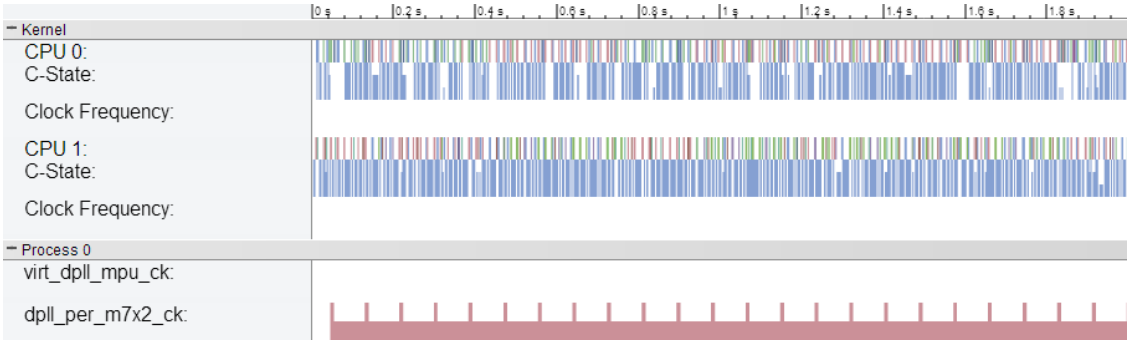
Figure 4.7: Pink Noise

Our low utilization audio test utilizes a pink noise audio file provided by Audio Check [40]. This pink noise runs at 96 kHz that runs for 30 seconds in a continuous loop on the Google Play Music application [41]. Pink noise has the characteristics where power spectral density is inversely proportional to the frequency of the signal [42]. In practicality, this noise

appears in a variety of applications and ubiquitous in nature; which we purposed to simulate users playing audio on the device [42][43]. According to these results, the tuned and static Ondemand governors both consume similar amounts of energy, but the Interactive governor consumes 9% to 4% more energy than the Ondemand governor. This is primarily due to cores constantly waking from idle and the 20 millisecond `sampling_rate` for the Interactive governor on both devices. Furthermore, we conclude this via traceview, seen in Figure 4.8, by the lack of blue, which represents the C-state or idle state during the execution of this test on the Galaxy Nexus.



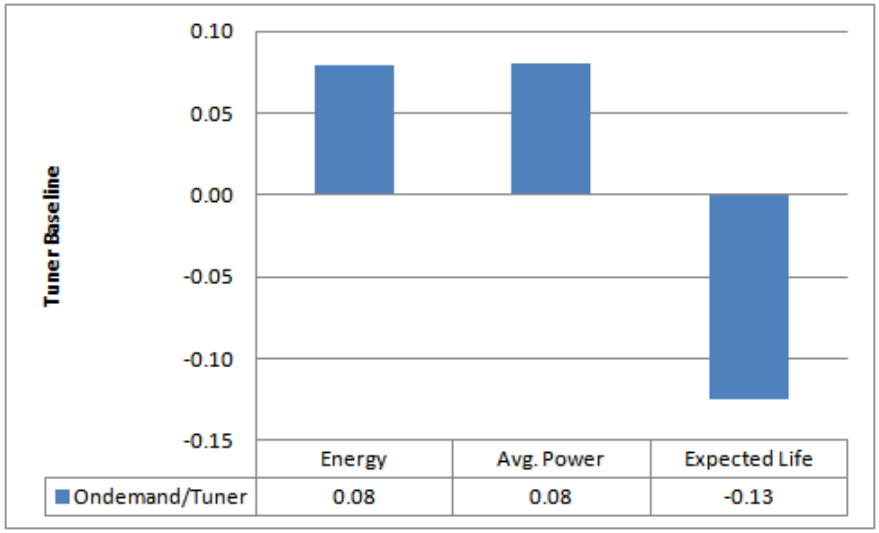
(a) Interactive Music Playback



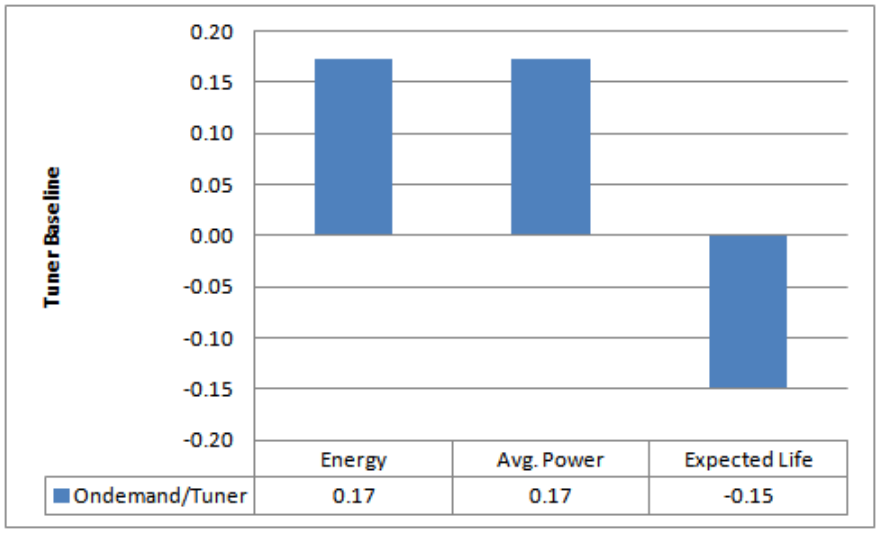
(b) Ondemand Music Playback

Figure 4.8: Music Traceview

Idle



(a) Nexus 4 Ondemand Idle



(b) Galaxy Nexus Ondemand Idle

Figure 4.9: Ondemand Idle Comparison

This test identifies if the `powersave_bias` has the ability to consistently reduce power consumption. The active applications during the test were the Android system services, such as the Google Play Service, with Wi-Fi enabled, and the screen off. We set the `powersave_bias` value to 400, which means reduce the current frequency by 40% of the target frequency that is allowed by the hardware frequency table (i.e. if the target frequency is 1.5 GHz the governor

will try to select a frequency close to 600 MHz). Adjusting the `powersave_bias` parameters allows the Ondemand to effectively lower power the energy consumption but still allow the system to scale the CPU frequency when background activities requires a higher frequency step. We assume that background tasks do not require scaling to the highest frequency because users have the screen off, but still need some performance to respond to calls or other important notifications. By scaling down from the maximum frequency, it shows that it is possible to save power for such devices at this type of scenario. This configuration does not impact the system performance when user's interacts with the device, because the tuner will dynamically reset the `powersave_-bias` parameter once the screen turned on.

Summary Analysis

The tests we have taken to compare our approach in the high, mid, and low utilization states to observe the effects of tuning the Ondemand settings and frequency steps.

From high utilization results, we see that the job scheduler has an important effect to the power consumption of the system. This occurs due to how the `cpufreq` governor evaluates the CPU utilization. The cost of reevaluating frequency once the core switched results in performance loss and higher energy consumption. In the Nexus 4 and Galaxy Nexus, both high utilization states our tuner shows that increasing the min frequency will have an impact in reducing the power consumption and improving performance.

In the mid utilization results, we see mixed results from the real and micro benchmarks. Although, our tests in Antutu and Angry Birds show potential energy savings without performance loss, our modified MiBench benchmark showed a different aspect for mid utilization. To clarify this difference, we performed a static sampling rate sweep of the MiBench tests, seen in Table 4.5. The lower the `sampling_rate` showed that it is possible to reduce average power usage, but results in lower performance. This is different from our tuner's

results, because the system takes into account of the variance in utilization and adjusts the min frequency and `up_threshold` values. By adjusting the `up_threshold` value, we allow the Ondemand governor to pick a frequency step that is not the lowest. We see that higher sampling rate will help for these types of repetitive busy to idle type of tests, but it correlates to higher energy consumption because of the performance benefits.

Nexus 4 FFT Sample Rate (µs)	50,000	100,000	150,000	200,000	250,000
Energy (µAh)	854.12	827.23	831.97	787.63	784.91
Avg. Power (mW)	1,230.24	1,208.30	1,169.76	1,126.48	1,123.00
Expected Battery (hr.)	6.49	6.60	6.82	7.08	7.10
Num. Tests	101.13	88.50	80.60	66.20	67.77
Nexus 4 Basicmath Sample Rate (µs)	50,000	100,000	150,000	200,000	250,000
Energy (µAh)	890.43	916.93	927.43	907.06	894.63
Avg. Power (mW)	1,312.68	1,302.25	1,287.64	1,280.73	1,275.87
Expected Battery (hr.)	6.07	6.12	6.19	6.23	6.25
Num. Tests	30.80	30.00	29.20	28.60	28.20

Table 4.5: MiBench Sample Rate Sweep

Finally, the low utilization results, we see that the Ondemand governor excels in conserving energy compared to the Interactive governor. We can see that the cores do not spend much time on idle in the Interactive governor, which resulted in higher energy consumption. Our approach with `powersave_bias` allows the system to avoid scaling to the highest frequency during a low utilization state. This is useful when the screen is off, since background tasks do not require scaling the frequency to the highest in this state. Therefore, we do reduce the performance during this state, but still able to retain performance when the screen is on and when the CPU utilization does not match this particular low utilization region.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we introduced the approach of tuning the `cpufreq` governor from the application layer, implemented with the Ondemand governor. We approached this problem at the application level to allow users to control the CPU governor to effectively configure their system to be on an application basis and by the system's utilization pattern. Unlike other methodologies that are only compatible with specific devices or by static profiles of the running foreground applications, this approach allows users to tune their existing system based on the utilization patterns and for each application. Since static profiles do not address potential idle utilization (i.e. user leaves the application running), this approach will dynamically tune the `cpufreq` governor for such utilization pattern. We compared our approach against the default Ondemand and Interactive governor settings on a Galaxy Nexus and Nexus 4 device. Our experiments show that even though the Ondemand governor is the same, each smartphone manufacturer has a different configuration that directly impacts the performance and energy consumption of their manufactured device. Since smartphones are

always on, setting the DVFS governor to a static setting is not beneficial for the system to improve its energy consumption. By Changing the governor tunables and frequency based on the CPU and system information, we allow the system to gain potential performance benefit and opportunity to conserve battery.

Our methodology of partitioning the CPU frequency average allows the system to categorize the utilization characteristics of the running application. By doing so, we are able to use a set of rules to tune the `cpufreq` governor to be more responsive and some cases conservative battery life based on the application's utilization characteristics. This is helpful since previous research show that the difference in power management governors also affects the latency of the system [23]. By tuning the governor, with application context, we can enable the `cpufreq` governors, such as Conservative or Ondemand to be responsive to certain utilization patterns that are running on the Android application layer, while enable energy savings at Low utilization states. For the Ondemand governor, our approach allows users to tune the governor to respond better for applications in both high and low utilization patterns.

Our approach highlights two important issues with current mobile multi-core systems. The first issue is the CPU scheduler in Linux. By allocating tasks to an idle core, there is no information passed from the original core to the idle core. This results in performance loss and increase power consumption due to incorrect operating frequency.

The second issue is the importance of varying the sampling rate and threshold values. By exploiting this fact on per-application basis, the system can consume less energy while performance degradation will not be noticeable. Since the sampling rate affects how often the governor decides when to scale the frequency from its previous sampled evaluation, it contributes to the frequency the CPU frequency switches between the frequency steps. As seen in the MiBench benchmarks, adjusting the right sampling rate and threshold for the system can help improve energy consumption while minimizing performance loss. We also see that

the higher sampling rate means that the system is sensitive to short utilization bursts in that occurs when the system is working on background processes, which results in increased energy consumption when the user is not actively interacting with the device.

5.2 Future Works

From this research, we see that there are opportunities to improve the performance and energy consumption even at the application level. The immediate improvement to our approach is to move the tuning system to the NDK level, to reduce the overhead cost from the Java layer. Although our test shows that we do not fully address the mid-utilization tests, the primary reason is because we are still using static configurations for each region category. We propose for future work that the tuner can propose self-determined policies to the users on an application basis.

We also see that the job scheduler could be improved by extracting information about the tasks that are in the cores work queue. Integrating the information provided by the scheduler, the `cpufreq` governor can switch to tasks identified as requiring a certain frequency. This is important because due to Linux Completely Fair Scheduler tasks migration results in different cores working in the non-optimal frequency [45]. In current `cpufreq` governor the high `sampling_rate` helps offset this performance bottleneck, but results in increase power consumption for tasks that requires a higher frequency. By integrating the scheduler and `cpufreq` governor via a kernel module or some higher level of abstraction, the idea of tuning the system is more effective because can proactively determine the optimal frequency for the active running tasks in the CPU work queues.

From this research, it shows clearly the importance of a tuning system with a power model to the existing mobile systems. Our tuning approach will be more efficient to include a

power model that provides information of the energy cost of changing the governor's tunable parameters. By including power models, such as Power Tutor [44], we can correlate the utilization load to the energy usage, temperature, and screen status to determine more accurate tunable settings that does not reduce performance and improve energy efficiency. With a model, we can determine the limitations of the tuning system and find exterior parameters, such as temperature that will affect the performance of the system. In short, to exceed the next energy savings overhead in the existing approach, it requires a higher dynamic tuning that abstracts our approach into the system layer and able to adjust the `cpufreq` governor settings and other sensors that are accessible at the kernel level. Mobile device users, will state their desire for performance or energy saving and in turn the tuning system should adjust the `cpufreq` governor and other options available by the Android framework and Linux kernel to achieve this balance.

Bibliography

[1] R. van der Meulen and J. Rivera, "Gartner says smartphone sales grew 46.5 percent in second quarter of 2013 and exceeded feature phone sales for first time." http://www.gartner.com/newsroom/id/2573415, 2013.

[2] S. (Stericson), "Busybox." "https://play.google.com/store/apps/details?id=stericson.busybox".

[3] V.-V. Helppi, "Android beyond the handset." "http://www.techdesignforums.com/practice/technique/android-beyond-the-handset/", apr 2010.

[4] N. G. Dominik Brodowski, "Linux cpufreq." "https://android.googlesource.com/kernel/msm/+/android-msm-hammerhead-3.4-kk-r1/Documentation/cpu-freq/governors.txt", May 2012.

[5] S. Datta, C. Bonnet, and N. Nikaein, "Android power management: Current and future trends," in *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pp. 48–53, June 2012.

[6] 3c, "Android tuner." "https://play.google.com/store/apps/details?id=ccc71.at".

[7] K. Mobile, "Battery doctor (battery saver)." "https://play.google.com/store/apps/details?id=com.ijinshan.kbatterydoctor_en", 2013.

[8] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica, "Collaborative energy debugging for mobile devices," in *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 2012.

[9] SINO, "No-frills cpu control." https://play.google.com/store/apps/details?id=it.sineo.android.noFrillsCPU".

[10] M. Huang, "Setcpu for android." "http://www.setcpu.com", 2013.

[11] "Adding features to your kernel." "http://xda-university.com/as-a-developer/adding-features-to-your-kernel".

[12] "About." http://www.cyanogenmod.org/about, 2013.

[13] stempox, "Cpu governors explained." "http://forum.xda-developers.com/showthread.php?t=1736168", June 2012.

[14] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[15] B. Dietrich and S. Chakraborty, "Managing power for closed-source android os games by lightweight graphics instrumentation," in *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, (Piscataway, NJ, USA), pp. 10:1–10:3, IEEE Press, 2012.

[16] P.-T. Liang, Wen-Yew; Lai, "An energy conservation dvfs algorithm for the android operating system," in *5th International Conference in Embedded and Multimedia Computing*, 2010.

[17] P. T. Bezerra, L. A. Araujo, G. B. Ribeiro, A. C. d. S. B. Neto, A. G. Silva-Filho, C. A. Siebra, F. Q.B. da Silva, A. L. Santos, A. Mascaro, and P. H. Costa, "Dynamic frequency scaling on android platforms for energy consumption reduction," in *Proceedings of the 8th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM2HW2N '13, (New York, NY, USA), pp. 189–196, ACM, 2013.

[18] Y. Zhang, X. Wang, X. Liu, Y. Liu, L. Zhuang, and F. Zhao, "Towards better cpu power management on multicore smartphones," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, (New York, NY, USA), pp. 11:1–11:5, ACM, 2013.

[19] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, (New York, NY, USA), pp. 179–194, ACM, 2010.

[20] "Samsung galaxy nexus i9250." "http://www.gsmarena.com/samsung_galaxy_nexus_i9250-4219.php", 2014.

[21] "Lg nexus 4 e960." "http://www.gsmarena.com/lg_nexus_4_e960-5048.php", 2014.

[22] M. Chan, "[patch 3/4] cpufreq: New 'interactive' governor." "https://lists.linaro.org/pipermail/linaro-kernel/2012-February/001120.html", Feburary 2012.

[23] S. Kim, H. Kim, J. Kim, J. Lee, and E. Seo, "Empirical analysis of power management schemes for multi-core smartphones," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, (New York, NY, USA), pp. 109:1–109:7, ACM, 2013.

[24] Y. Song, "cpufreq: Add sampling window to enhance ondemand governor power efficiency." "http://lwn.net/Articles/420930/", 2010.

[25] "cpufreq_ondemand.c." "https://android.googlesource.com/kernel/msm.git/+/f2ea7e7e6bef40b2e93619699e35e6e419f1f026/drivers/cpufreq/cpufreq_ondemand.c".

[26] "cpufreq_ondemand.c." "https://android.googlesource.com/kernel/msm/+/android-msm-mako-3.4-jb-mr1.1/drivers/cpufreq/cpufreq_ondemand.c".

[27] "2.7 timing in the linux kernel." "http://www.6test.edu.cn/~lujx/linux_networking/0131777203_ch02lev1sec7.html".

[28] "Monsoon power monitor." "http://www.msoon.com/LabEquipment/PowerMonitor/".

[29] "Autoit - autoitscript." "http://www.autoitscript.com/site/autoit/".

[30] "Monkey runner." "http://developer.android.com/tools/help/monkeyrunner_concepts.html".

[31] E. Goslawski, "Repetitouch pro (root)," April 2014.

[32] "Linpack." "http://www.netlib.org/linpack/".

[33] R. Longbottom, "Roy longbottom's android benchmark apps free and easy with no ads." "http://www.roylongbottom.org.uk/android%20benchmarks.htm".

[34] "Neon." "http://www.arm.com/products/processors/technologies/neon.php".

[35] "Android ndk." "https://developer.android.com/tools/sdk/ndk/index.html".

[36] "Profiling with traceview and dmtracedump." "http://developer.android.com/tools/debugging/debugging-tracing.html", 2014.

[37] I. AnTuTu, "Antutu benchmark." "http://www.antutulabs.com/downloads.html", 2013.

[38] A. A. T. Team, "Fps meter." "https://play.google.com/store/apps/details?id=com.aatt.fpsm".

[39] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[40] A. Check, "High definition audio test files." "http://www.audiocheck.net/testtones_highdefinitionaudio.php".

[41] G. Play. "https://play.google.com/store", 2014.

[42] J. Castro, "What is pink noise?." "http://www.livescience.com/38464-what-is-pink-noise.html", July 2013.

[43] P. Bak, C. Tang, and K. Wiesenfeld, "Self-organized criticality: An explanation of the 1/f noise," *Phys. Rev. Lett.*, vol. 59, pp. 381–384, Jul 1987.

[44] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[45] M. T. Jones, "Inside the linux 2.6 completely fair scheduler." `"http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/"`, dec 2009.

Appendices

A Appendix A

This is the repository link for the project. There are three projects in this repo that contains the applications that were used for this thesis
https://bitbucket.org/linsw/androiddvfstuner

The following explains the projects that are in this repository.

BenchmarkNativeTest contains the modified mibench tests for our constant repetition tests
ProfilerAutomation contains the AutoIt3 scripts to control the benchmark tests and batch files used to call the MonkeyRunner scripts
ProfilerPhone is Android DVFS Tuner project for the Android devices.

B Appendix B

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,144.55	1,127.47	1,117.12	2.46%	0.93%
Avg Power (mW)	1,521.15	1,504.44	1,480.27	2.76%	1.63%
Mflops	163.00	163.42	160.99	1.25%	1.51%
Exe. Time (s)	7.80	7.83	7.73	0.92%	1.25%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,148.07	1,094.07	1,076.78	6.62%	1.61%
Avg Power (mW)	1,564.67	1,492.39	1,467.47	6.62%	1.70%
Mflops	249.78	249.73	246.92	1.16%	1.14%
Exe. Time (s)	7.81	7.61	7.41	5.39%	2.65%

Table B.1: Linpack NDK Tests

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (textmu Ah)	1,319.89	1,263.51	1,171.62	12.65%	7.84%
Avg Power (mW)	1,749.68	1,673.90	1,545.69	13.20%	8.29%
Mflops	48.03	61.42	51.13	-6.06%	20.12%
Exe. Time (s)	8.94	8.31	7.57	18.10%	9.78%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,178.45	1,151.56	1,152.17	2.28%	-0.05%
Avg Power (mW)	1,669.10	1,629.04	1,632.23	2.26%	-0.20%
Mflops	59.81	62.01	61.27	-2.38%	1.20%
Exe. Time (s)	7.66	8.47	7.56	1.40%	12.10%

Table B.2: Linpack Java Tests

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,182.62	1,169.67	1,107.40	6.79%	5.62%
Avg Power (mW)	1,570.36	1,549.98	1,470.41	6.80%	5.41%
Mflops	384.32	384.83	411.18	-6.53%	-6.41%
Exe. Time (s)	7.81	7.85	7.34	6.28%	6.93%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,295.13	1,306.00	1,287.96	0.56%	1.40%
Avg Power (mW)	1,725.38	1,745.28	1,727.12	-0.10%	1.05%
Mflops	823.99	822.95	813.00	1.35%	1.22%
Exe. Time (s)	8.11	8.21	7.82	3.69%	5.02%

Table B.3: Linpack NEON Tests

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (textmu Ah)	2,695.13	2,822.70	2,707.46	-0.46%	4.26%
Avg Power (mW)	1,204.96	1,263.79	1,215.94	-0.90%	3.94%
Expected Battery (hr)	6.66	6.40	6.61	0.74%	-3.23%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	2,374.07	2,520.93	2,346.46	1.18%	7.44%
Avg Power (mW)	1,080.76	1,162.26	1,068.09	1.19%	8.82%
Expected Battery (hr)	7.38	6.87	7.47	-1.22%	-8.08%

Table B.4: Audio - Pink Noise

Galaxy Nexus	Ondemand	Tuner	Ondemand/Tuner
Energy (µAh)	56.00	47.77	17.23%
Avg. Power (mW)	24.77	21.11	17.32%
Expected Life (hr)	292.25	342.90	-14.77%
Nexus 4	Ondemand	Tuner	Ondemand/Tuner
Energy (µAh)	86.29	79.92	7.97%
Avg. Power (mW)	39.22	36.31	8.01%
Expected Life (hr)	229.31	262.10	-12.51%

Table B.5: Idle - Screen off

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	30,376.83	31,648.86	31,040.41	-2.14%	1.96%
Avg. Power (mW)	2,143.53	2,229.32	2,159.34	-0.73%	3.24%
Score	9,990.31	10,218.00	10,414.27	-4.07%	-1.88%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	37,507.12	37,857.65	36,595.29	2.49%	3.45%
Avg. Power (mW)	2,930.20	2,957.64	2,859.10	2.49%	3.45%
Score	18,105.71	18,393.43	18,098.94	0.04%	1.63%

Table B.6: Antutu

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	6,400.48	6,824.69	6,281.79	1.89%	8.64%
Avg. Power (mW)	1,721.24	1,840.61	1,691.99	1.73%	8.78%
FPS	36.80	36.50	36.00	2.22%	1.39%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	6,588.61	6,761.29	6,200.31	6.26%	9.05%
Avg. Power (mW)	1,810.57	1,856.04	1,724.67	4.98%	7.62%
FPS	48.10	48.20	47.30	1.69%	1.90%

Table B.7: Angry Birds

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	967.11	1,076.09	1,013.03	-4.53%	6.23%
Avg. Power (mW)	1,201.28	1,322.53	1,277.54	-5.97%	3.52%
Total Test (s)	4.96	3.67	4.88	1.64%	-24.78%
Num. Tests	91.80	119.00	93.78	-2.11%	26.90%
Avg. Test Time (s)	0.05	0.03	0.05	5.22%	-40.13%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	870.24	976.45	978.79	-11.09%	-0.24%
Avg. Power (mW)	1,233.24	1,266.46	1,246.67	-1.08%	1.59%
Expected Life (hr)	6.47	6.30	6.40	1.09%	-1.50%
Total Test Time (s)	4.46	3.22	3.55	25.60%	-9.51%
Num. Tests	101.20	124.80	114.60	-11.69%	8.90%
Avg. Test Time (s)	0.04	0.03	0.03	42.23%	-16.88%

Table B.8: Mibench FFT

Galaxy Nexus	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	1,015.10	1,050.66	1,097.65	-7.52%	-4.28%
Avg. Power (mW)	1,369.06	1,333.65	1,381.13	-0.87%	-3.44%
Total Tests Time(s)	5.41	5.25	5.07	6.61%	3.50%
Num. Tests	30.56	32.00	32.13	-4.89%	-0.39%
Avg. Test Time (s)	0.16	0.13	0.16	-2.92%	-19.54%
Nexus 4	Ondemand	Interactive	Tuner	Ondemand /Tuner	Interactive /Tuner
Energy (µAh)	890.43	1,006.59	1,035.81	-14.04%	-2.82%
Avg. Power (mW)	1,312.68	1,279.34	1,321.58	-0.67%	-3.20%
Expected Life (hr)	6.07	6.23	6.03	0.70%	3.32%
Total Tests Time(s)	5.39	5.25	5.07	6.23%	3.55%
Num. Tests	30.80	32.00	32.20	-4.35%	-0.62%
Avg. Test Time (s)	0.17	0.16	0.16	11.06%	4.17%

Table B.9: Mibench Basicmath

Nexus 4 FFT Sample Rate	50,000	100,000	150,000	200,000	250,000
Energy (µAh)	854.12	827.23	831.97	787.63	784.91
Avg Power (mW)	1,230.24	1,208.30	1,169.76	1,126.48	1,123.00
Expected Battery (hr)	6.49	6.60	6.82	7.08	7.10
Total Tests Time(s)	4.46	5.03	5.42	6.08	6.03
Num. Tests	101.13	88.50	80.60	66.20	67.77
Avg. Text Time (s)	0.04	0.06	0.07	0.09	0.09
Nexus 4 Basicmath Sample Rate	50,000	100,000	150,000	200,000	250,000
Energy (µAh)	890.43	916.93	927.43	907.06	894.63
Avg Power (mW)	1,312.68	1,302.25	1,287.64	1,280.73	1,275.87
Expected Battery	6.07	6.12	6.19	6.23	6.25
Total Tests Time(s)	5.39	5.59	5.67	5.74	5.80
Num. Tests	30.80	30.00	29.20	28.60	28.20
Avg. Text Time (s)	0.17	0.19	0.19	0.20	0.21

Table B.10: Mibench Sample Rate Sweep