MOBICORE: AN ADAPTIVE HYBRID APPROACH FOR POWER-EFFICIENT CPU MANAGEMENT ON ANDROID DEVICES

by

Lucie Broyde

B.S. in Electrical Engineering, ESIGELEC, 2015

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Lucie Broyde

It was defended on

March, 15th 2017

and approved by

Yiran Chen, Ph.D, Associate Professor
Department of Electrical and Computer Engineering

Zhi-Hong Mao, Ph.D, Associate Professor
Department of Electrical and Computer Engineering

Amro El-Jaroudi, Ph.D, Associate Professor
Department of Electrical and Computer Engineering

Thesis Advisor: Yiran Chen, Ph.D, Associate Professor
Department of Electrical and Computer Engineering

MOBICORE: AN ADAPTIVE HYBRID APPROACH FOR POWER-EFFICIENT CPU MANAGEMENT ON ANDROID DEVICES

Lucie Broyde, M.S

University of Pittsburgh, 2017

Smartphones are becoming essential devices used for various types of applications in our daily life. To satisfy the ever-increasing performance requirement, the number of CPU cores in a phone keeps growing, which imposes a great impact on its power consumption. This work presents a series of analysis to understand how the current Android resource management policy adjusts CPU features. Our results indicate a significant improvement margin for CPU power efficiency in modern Android smartphones. We then propose MobiCore – a power-efficient CPU management scheme that can optimize the use of Dynamic and Frequency Voltage Scaling (DVFS) and the Dynamic Core Scaling (DCS) techniques with a sensitive control on CPU bandwidth. The measurements on the real systems prove that MobiCore can achieve substantial CPU power reduction compared to state-of-the-art architecture.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE - ACKNOWLEDGEMENT

I started my master degree as a third-year student from my engineering school ESIGELEC back in France. This year and a half was the most painful and meaningful time of my life. I left my original specialization which was networks and telecommunications to switch to something I thought I will never do again: Electrical and Computer Engineering and more specifically software and hardware design, embedded system and computer architecture. I didn't know what was "doing research". I hated having issues and spend days solving them, I was here now and no matter what I had to "do something". It was frustrating seeing all those people speaking about research, results, raising questions and knowing what tool to use to solve them. I didn't have those automatisms and I tried to learn how to do research on my own. This was the most painful thing I ever did, I had to understand and show people that I could be part of the "team". There is nothing more delightful than having the feeling you are doing something useful and that somebody cares about your progress. I learned a lot about myself by suffering but I think it was worth it. I discovered new people, new systems and new opportunities.

I want to thank my mother for her support to my long lamentation e-mails. My father for his ideas and suggestions.

Thanks to Dr. El-Nokali who brought me here. For his support and availability for my endless questions. For his trust in me as well as for the great opportunity he offered me.

My advisor, Dr. Chen giving me the opportunity to do research on his behalf. As well as for the time he spent for me.

Sandy Weisberg for her constant support for all international students.

1.0 INTRODUCTION

Due to battery constraints, energy efficiency is, today, the main concern in mobile devices, such as the smartphone. Given the decreasing size of hardware components, technology helps us to combine more and more components together in a smaller environment. The emerging market for multi-core processors is giving new opportunities in terms of resources management as we have more options in our hand such as variability which enables us to get the exact output from those hardware components. From single core architecture, we went through two-core, four-core, octa-core and now reaching deca-core implementation in modern mobile devices [17]. Such new systems and architectures raise new challenges and new opportunities for energy consumption and performance level.

Multi-core processors are now common design in embedded systems and they add a whole new dimension to the given opportunities for management in mobile devices. It now becomes essential for mobile applications to fully exploit the potentials of this available embedded hardware resource [15]. Especially for the ever increasing high-demand in computing performance in mobile devices which is going higher each year [15]. Evolution of multicore architectures helped reaching those requirements, however, also results in severe power consumption issu when many cores are active simultaneously [2]. Besides that, it may appear that using more cores at the same time does not necessarily lead to better performance or better user-experience because of many realistic constraints [3]. Hence, there is significant room to

save power for mobile devices while maintaining good performance level by carefully managing the available processing cores [14].

1.1 BACKGROUND

In modern multi-core processors embedded in mobile devices, the OS system (here Android OS) provides multiple policies to control energy consumption. The first one is off-lining cores, also called dynamic cores scaling (DCS) which allows the operating system to switch off single cores, reducing per-core power consumption to almost nothing and maintaining the other cores at an active state. The second one is the well-known dynamic voltage and frequency scaling, also called DVFS in the literature, which enables the system to switch between frequencies and apply the just-needed voltage supply to get the determined frequency. This last technique is done at the expense of performance. These two techniques are respectively implemented as *hotplug* and *governors* policies in the Android-Linux architecture. Both techniques are based on CPU workload. This value can be retrieved from the Linux architecture by reading some *log* files and then the OS can dynamically adapt the allocated resources each period of time *t.* Basically, if the workload is increasing, we will increase/decrease the frequency of cores or increase/decrease the number of active cores. Implementing and using these two mechanisms in tandem does present interesting values for power consumption and performance. The optimal decision will be taken in consideration of the best power-performance ratio results of each of the two mechanisms. The scheme giving the best one is chosen by the Android system.

But these two schemes are neither unified nor coordinated in the real implementation as they both have two different interfaces in the Linux architecture [14]; that shows that there exist

considerable improvements that could be done for more power efficiency and performance level if one can use them at the same time and let them complement each other.

1.2 MOTIVATION

The rapid growth of computation performance requirement motivates smartphone manufacturers to integrate more and more CPUs on mobile devices. However, such a practice also results in considerably high power consumption. To validate this observation, we ran experiments and stressed (means that we tweaked CPU features such as CPU utilization, frequency or activation) the CPU cores on multiple phones released between 2010 and 2014 with an in-house kernel app (more details about this app will be given in Section 3). The tested phones, i.e., Samsung Nexus S, Motorola mb810, Samsung Galaxy S II, LG Nexus 4, Nexus 5, LG G3, are using different number of CPU cores, as shown in Figure 1. This figure also shows that the total power consumption of the phones increases almost linearly with the number of CPU cores. Among the phones with the same number of CPU cores, the total power consumption of the new phones is also slightly higher than that of the old ones showing the growth of the computing power consumption of a single core itself.

To understand how significantly the computing power contributes to the total power consumption of the smartphones, an infrared picture of two Android phones taken by an infrared sensor [1] is depicted in Figure 2(a). The left one is a Samsung Nexus S with only one CPU core while the right one is a Nexus 5 with four identical CPU cores. All the phones are running at their highest computing state using the same kernel application. On both phones, we can easily identify where the CPU cores are located as these areas reach temperature levels clearly higher

(clearer in the picture) than the others: the CPU area of the Nexus 5 reaches 42.1°C while the one of the Nexus S reaches 26.9°C. That obviously shows that the multicore architecture on the Nexus 5 is much more power-hungry than the single-core architecture on the Nexus S. In summary, we have seen that more cores are more power hungry and that recent cores themselves are more power hungry because of the better performance level they can reach. Note that the total average power consumptions of the Nexus S and Nexus 5 are 2403.82 mW and 980.6 mW, respectively. The Nexus 5 is 140% more power consuming than the Nexus S.

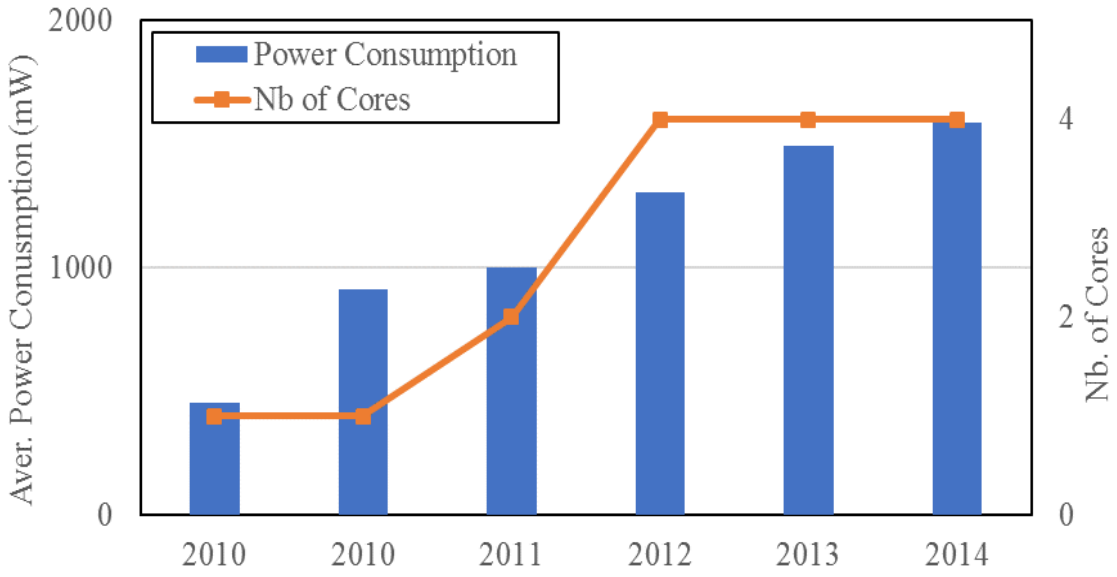
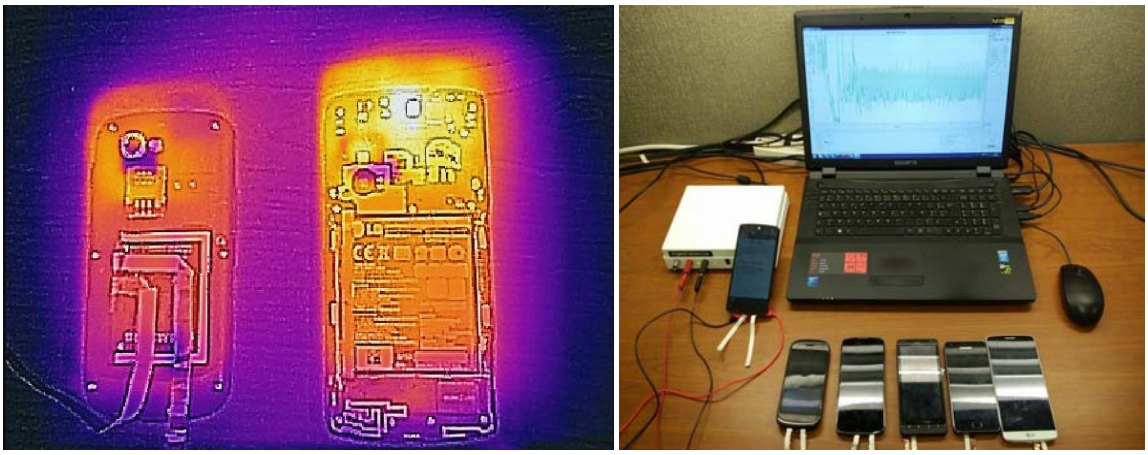


Figure 1. Evolution of average power consumption for different phones



(a) IR image **(b)** Experimental Setup

Figure 2. Experimental measurements

The two above experiments show that there is some improvement to do in terms of usage of hardware resources when speaking of multi-core architecture design. Having a higher temperature for a multi-core architecture mobile device makes sense as there are more hardware components involved. But our experiment shows that giving the whole resources blindly is not going to work on new architectures. Indeed, it will consume too much power and despite achieving a good (or even very good) performance, it will not last long as the temperature will be very high and this will damage hardware components (being one of the first major limitations). Having more components and more available resources imply a good resource management so that we can achieve some power savings, good performance, and make the components live longer as the temperature will be limited. Indeed, if a good resource management is made, hardware components will end up being less used, lowering the temperature level, decreasing power consumption (as power and temperature are proportional) and this leads to a longer life span for the mobile device.

1.3 PREVIOUS WORK

Originally, the Android default policy [7], [8] applies either DVFS or DCS according to the workload. MobiCore is merging the two above policies to enable a more fine-grained algorithm and decision making. Many studies are taking into consideration some types of information. For example, application aware policies [9] design a new governor based on application characteristics but do not consider reevaluating the number of online cores although they say that usually the *hotplug* policy doesn't make the right decision. Task-scheduling algorithms have also been studied: [10] takes advantages of the multicore architecture to better schedule the work among them but don't consider setting a new frequency to each core in response to the task's priority so that more power savings can be obtained; [11] is also re-scheduling tasks among the cores but when it comes to user-centric task, the highest frequency is automatically chosen and results in wasting power. Moreover, this work has only been simulated on a Linux based machine. A thread sensitivity model is presented in [12]. This work considers three important sensitivity states for threads and modifies the allocated CPU resource according to these states.

However, for a heavy workload it automatically chooses the highest level which may waste power as well. All of those works present some good solutions to complement DVFS and DCS but most of them are not able to react fast enough to sudden change in workload or to heavy computing demand. Thus, they cannot be adaptive enough to the specific dynamicity of games. The idea is a new design more adaptive to the variation of workload so that its algorithm finds the best combination between the number of cores and the frequency allocated for each of those cores.

1.4 CONTRIBUTION OF THIS THESIS

As stated before, our work in this thesis will be to try to bridge the obvious gap that exists in the Android system. Our goal is to get the best user experience so in other words get the best performance and to consume as few power as possible. For that purpose, we will work on a new type of *scheme* which is going to combine the action of the DVFS and the DCS to get a more precise, accurate and power-efficient choice.

We will first perform a comprehensive analysis on the impact of the number of CPU cores and the core's working frequency on power consumption and performance of Android devices. Then, based on our analysis, we will propose MobiCore – a hybrid adaptive approach for CPU power management. According to the amount of workload to be processed, our optimized solution will determine the state (i.e., activation and frequency) of the CPU cores to be adopted. To get a more responsive algorithm to dynamic workload, we will analyze the variation of the workload to determine the computing need at the next time step and a scaling factor will be applied to reduce or expand the CPU bandwidth.

1.5 SUMMARY

This section introduced the Android architecture, the default policy in current Android devices and the existing gap in power consumption which can be filled up. We saw that this technique is still used today but is not suitable to new multicore architectures and this is why we are proposing a more adaptive scheme called MobiCore. Our major contributions in this work can be summarized as follows:

- We performed a quantitatively study on real Android mobile devices about the tradeoffs between changing the number of cores and their frequencies when CPU workload varies;
- Based on the above analysis, we proposed MobiCore: a new approach that can simultaneously optimize the number of active CPU cores and the working frequencies of each core along with controlling the allocated CPU bandwidth;
- We implemented MobiCore on a Nexus 5 smartphone and evaluated its effectiveness with different types of dynamic and more static applications.

2.0 ANDROID POWER MANAGEMENT POLICY

As we said before, the criteria which is used to determine how should be allocated the hardware resources is the CPU workload (also described as CPU utilization). We will first explain how a processor works and how we can tweak its state to modify the power consumption and we will define the CPU utilization, then we will explain how the different management policies are working and finally describe how should be design the new hybrid management.

2.1 HARDWARE - THE DIFFERENT STATES OF CPU

When it comes to the higher number of processors, we are speaking about more power consumption. Indeed, the largest the number of active processors, the more static power there is and that means more power consumption.

A processor has three different states: active, sleeping (idle) and off-line. The transition from one state to another is more or less long and the power consumption of each states is really different. The off-line state consumes almost nothing. It corresponds to the deepest sleep state in which power consumption is minimal (almost 0). Instructions are executed in the active state. Its power consumption depends on the frequency chosen. Basically, more power consumption is observed for a higher frequency. Whereas the idle state is the state in which the core is ready to execute instructions (but is not executing), it is a less-deep sleep and consumes more power that

the power consumed by the off-line state. Note that, this power consumption mainly depends on how the platform is powered.

Research has been done in the past showing the issue of the increasing power consumption [18], [19] and [21]. Due to the larger number of processors, it has been found that the possible power reduction of switching processors states is higher than the power reduction obtained by only DVFS [20]. That shows that if power reduction is the main point of a new method, switching between CPU states have to considered along with the DVFS. From past research mentioned above, consequent power reduction can be achieved when the off-line state is chosen. Idle state does not bring enough power reduction. That means that we do not consider anymore the possible race-to-idle principle benefit, well explained in [24]. To achieve substantial reduction in power consumption, an off-lining policy replacing the race-to-idle design and a minimum per-core frequency policy shall be chosen.

2.2 SOFTWARE - DEFAULT MANAGEMENT CPU UTILIZATION

CPU utilization (or CPU workload) is defined as the percentage of CPU cycles that is needed for a computation task. For the multi-core scenario, the overall CPU utilization is defined as the average of the utilizations over all the CPU cores. The two default power management policies in Android Linux based architecture – DVFS and DCS, all use CPU utilization as an important index in their management schemes. The basic principle is to fairly allocate the available CPU resources and to balance the workload among cores to achieve an ideal multitasking. The two approaches will be described individually.

2.2.1 Governors or Dynamic Voltage and Frequency Scaling (DVFS)

The DVFS is a basic principle which says that a specific frequency needs a specific minimum voltage. That method can effectively achieve power reduction. In the Android-Linux OS architecture, we can choose the *governor* which is going to manage the frequency of the cores depending on the CPU workload. The default governor on Android system is called *ondemand*. Briefly, the *ondemand* governor checks the workload given to a CPU core at the current time. If the load reaches a set frequency threshold, CPU frequency raises to the maximum frequency. Same behavior for decreasing workload. It is the most reliable governor as it can quickly respond to a high-demand in computing performance. Currently, there are six different governors in the Linux architecture. There is the *interactive* governor based on the current workload as the *ondemand* governor. It is used for latency-sensitive workloads. However, it has a much more aggressive CPU speed scaling in response to the CPU activity. The *conservative* governor which is also based on the current usage but it increases (decreases) the CPU speed more smoothly (instead of suddenly jumping to the highest frequency). This one is more suitable for a power-friendly environment. The *powersave* governor which is given two frequency thresholds and chooses the minimum frequency between those two thresholds. The *performance* governor which is working the same way as the *powersave* one but sets the highest frequency between two frequency thresholds. Finally, the *userspace* governor is here for users who want to try their own hand-written governor. In summary, all governors have a specific way of allocating hardware resources depending on the computation need [23].

The approach of the *ondemand* governor based on the CPU workload, having the load threshold and giving the highest frequency when there is a sudden burst in workload ensures to deliver the needed CPU resources to make up with the performance when needed. This technique

is suitable for every type of applications but is not a battery-powered friendly governor for high-computing applications such as games. First, this policy was introduced a long time ago and originally designed for the single-core mobile architecture [7]. Second, it can obviously be improved if considering a better workload sensitive approach which instead of giving the highest possible frequency will give the just-needed frequency thus saving some power. In conclusion, improvements are possible in terms of a more-accurate DVFS and also by considering the many possibilities of the multi-core architecture.

2.2.2 Hotplug or Dynamic Cores Scaling (DCS)

To obtain more control on a multicore architecture, we need to manage the set of active hardware. For that purpose, the *hotplug* policy has been introduced in the Linux architecture. *Hotplug* enables the kernel to dynamically activate more or less hardware components [27].

mpdecision is a service which protects the phone from turning off cores. In order to be able to activate that feature, we need to inactivate the *mpdecision* service by sending a simple command through the terminal with *adb shell*. The default *hotplug* policy will now be able to off-line cores. This policy allocates the hardware resources depending on the amount of workload. Basically, more cores for a high workload and less cores for a low workload. This method can effectively reach the performance requirements from the running process but for this end will not prioritize a power saving scheme. Indeed, the choice is not precise enough; it is either activate or inactivate cores which is a little abrupt. Then, alone it cannot be efficient for a trade-offs scheme in any way because only the activation of cores is chosen and not their effective speed.

2.3 THE IDEA BEHIND MOBICORE

The default policy of the Android system has been created to achieve the best power savings/performance trade-off for a basic usage of the phone using DVFS or DCS alternatively. It is giving good results for dynamic and static workload. But there does not exist a systematical guidance or even a mechanism for the designer to apply these two policies at the same time to achieve both power saving and adequate performance. As both decision makers are based on CPU workload, the idea is to find a solution which combines the number of activated cores and the exact frequency for each core. There must exist an ideal combination of number of cores and frequency. This operating point must be the one chosen by our new policy called MobiCore.

2.4 SUMMARY

The different states of the CPU processor have been described in this section. We need to take advantage of the three CPU states and their different characteristics in terms of power consumption, reaction time, etc. Besides this, we saw that the Android default policy is equipped to adapt either the frequency or the number of cores but that these decisions are independent and thus there are not making any common arrangement. In summary, we can change the “speed” (aka. frequency) and the number of “motors” (aka. cores) by carefully analyzing the required “acceleration” (aka. workload). We understood that the idea is to make up with a solution unifying DVFS and DCS. The idea is that: for each variation of workload, an optimal operating point is to be found.

3.0 EXPLORING THE TRADE-OFF IN THE ANDROID MANAGEMENT

In this section, we will try to find out how the power consumption is affected by the change in frequency along with the number of cores. Experiments, results and analysis will be presented. Thus, we hope to find some correlation to get a solution which finds optimal working points.

3.1 EXPERIMENTAL SETUP

For the purpose of our experiments, we used an in-house kernel application to measure the influence of the CPU utilization on the mobile phone power consumption when changing those hardware features. This application is characterized by configurable busy loops which do not include any memory accesses. The load is going on for a certain number of iterations and includes a period of idleness, which is about 40ms. This application allows us to change the number of active CPU cores, the allowed overall CPU utilization and the frequency of each core. This altogether makes it possible to tweak and stress different CPU features. Running it in the background produces a file recording historical information of the hardware states.

The phone used in our analysis is a Nexus 5 with the following characteristics [4]. It has a Snapdragon 800 chipset with quad-core CPU. The four identical CPU cores can work at 14 different frequencies ranging from 300MHz to 2.2656GHz. The phone has been rooted and is running a clean version of Android 6.0 OS. For more details see Table 1 below. For power

measurements, we used a power meter named Power Monsoon externally connected to the mobile device [16]. The battery of the phone has previously been removed and power consumption is measured directly at the power pins. The experimental setup is shown in Figure 2(b). The airplane mode is enabled and the screen is turned off to eliminate the effects of power consumption on communication and display components. The default Android policy is setup. Two low, two high, and one middle frequencies have been chosen to be benchmarked as they represent the wide variety of the available frequencies.

Table 1: Specifications of the Nexus 5 platform [26]

Specifications	Mobile devices: Nexus 5
SoC	Snapdragon 800 (MSM8974)
CPU (4)	Quad-core 2.3 GHz Krait 400
Freq. min: 300MHz	
Freq. max: 2 256GHz	
Volt. min: 0.9V	
Volt. max: 1.2V	
GPU	Adreno 330
Freq. max: 450MHz	
Cache (L2)	2048 (kilobytes)
OS	Android 6.0 (Marshmallow)

3.2 CONSTRAINTS IMPOSED

In this section, we will go through the different assumptions and constraints that we made for our analysis. Indeed, a mobile device is a system which embeds many different types of components (CPU, memory, GPU, etc). Those un-core components are affecting the behavior of the system

such as the execution time, the graphics rendering, the power consumption, the scheduling steps for tasks and more.

When processing a workload, the Linux architecture uses a task scheduler to process the task according to a degree of prioritization. The default Linux task scheduler is splitting the workload over a certain number of processes (here 4). This is almost not affecting the work performed by each core involved in the benchmark. After some analysis and according to [12], we can report an average of three to four iterations of the workload with a relatively small standard deviation.

One will say that the GPU is also a bottleneck in terms of graphic processing. In order to minimize this bottleneck, we chose to set the GPU frequency to the highest one so that ideally and theoretically the GPU processes any requests from CPU cores as quick as possible. This is to minimize its influence on the running process and to avoid any variation in terms of power consumption. Of course, it will affect the power consumption but as it will be stable we will be able to remove it from our measurements.

Concerning the memory bandwidth, it will be setup to the highest. By default, we can switch from one low to one high frequency; the highest frequency is always chosen when an application is launched as it needs full throughput from memory to avoid any bottleneck.

3.3 COST OF BUSY CYCLE WITH FREQUENCY & CORES

3.3.1 Effect along with frequency

In this section, we will try to figure out how a local CPU load affects the power consumption of the phone along with the frequency. We characterized the effect of one active CPU core using the kernel application during 1-min period of time

In Figure 3, we can see the influence of the CPU utilization for one single core for five different frequencies on the phone's power consumption. We observe that increasing the CPU load from 10% to 100% increases the power consumption by up to 74% at the highest frequency and 62.5% at the lowest frequency. At 100% CPU utilization, scaling frequency down to the lowest frequency brings power savings from 28.2% up to 71.9%. This shows us that the energy consumed is an increasing function of frequency for a single core; in other words, the lowest energy consumed is reached when the lowest frequency is chosen.

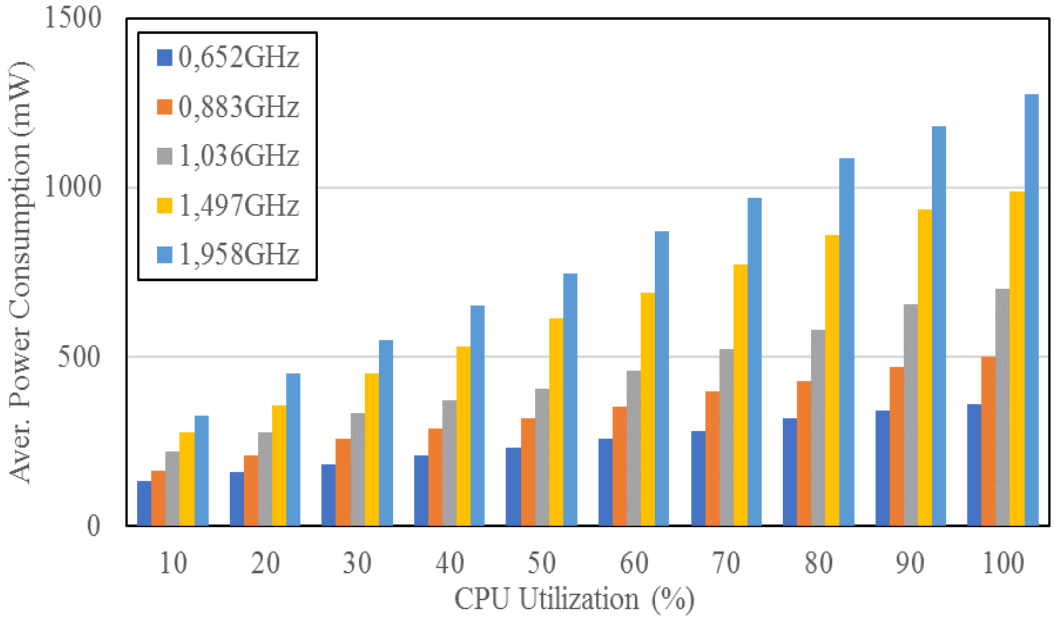


Figure 3. Pow. cons. over CPU utilization at different freq. for 1 core

3.3.2 Effect along with number of cores

In this part, we will prove that adding more active cores increases the power consumption. We used the same kernel application to test the influence of the number of active CPU cores on the phone's power consumption. The local CPU utilization is fixed at 100% for all cores to avoid any effect from busy cycle's bottleneck.

Figure 4 shows the results at five different CPU frequencies. Interestingly, the power consumed is not a linear function with respect to the number of active cores (i.e., it does not scale proportionally when going from 1 to 2 cores or from 2 to 4 cores). In fact, for the same frequency, i.e., the highest one, going from 1 core to 2 cores leads to an increase of 28.3% in power consumption and of only 7.7% from 2 to 4 cores. For a lower frequency, we have an increase of 17.3% and of 6.4% respectively. Going from 1 to 2 cores is aggressive in terms of

power consumption and going from 2 to 3 cores or from 3 to 4 cores is more beneficial due to the marginal power increase. But increasing the frequency for 1, 2, 3 or 4 cores considerably affects the power consumption by up to 70% approximately for each case. After observation, we can state that for a fixed frequency, the power consumption is highly dependent on the number of online cores. Note that the difference varies sometimes by a factor of two.

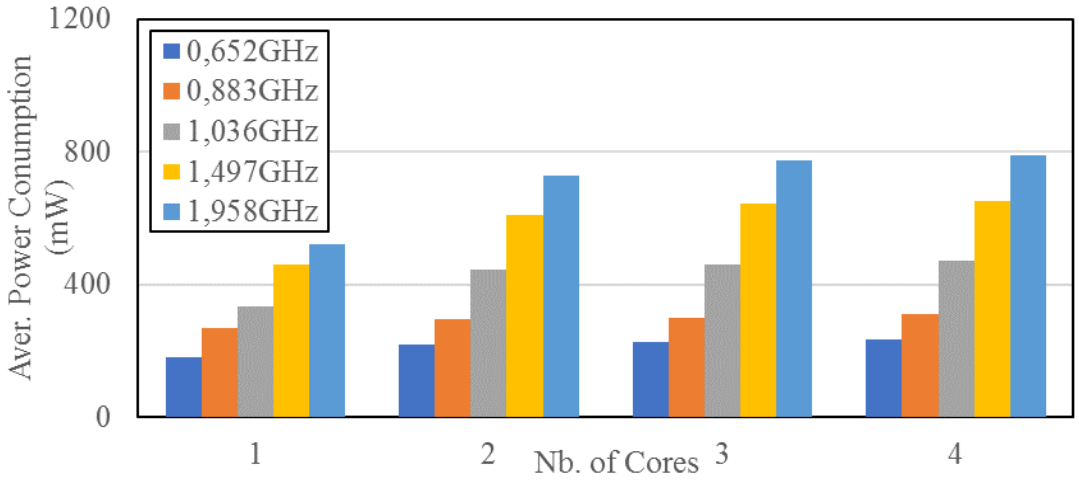


Figure 4. Pow. Cons. over CPU cores for different freq. at 100% CPU utilization

3.4 FINDING AN OPTIMAL OPERATING POINT

In this section, we will try to figure out what are the different operating points of the phone. An operating point is defined by a certain amount of hardware resources including their features. Here we are speaking about number of online cores along with their individual frequency. To separate those operating points, we will take another criterion: the global CPU load. Briefly, to achieve a certain amount of global utilization, there is a minimum amount of hardware resources

required. Starting from this point, we can test out all the remaining possible combinations above this threshold. For example, a 100% global CPU load needs all the core active at their highest frequency. If we apply this rule we can find different CPU combinations in terms of number of active cores and their active frequency (aka. operating points). We will then have for 100% global CPU load, all running cores at their maximum frequency, for 50% CPU load, all the cores at half frequency or two cores at their maximum frequency, etc.

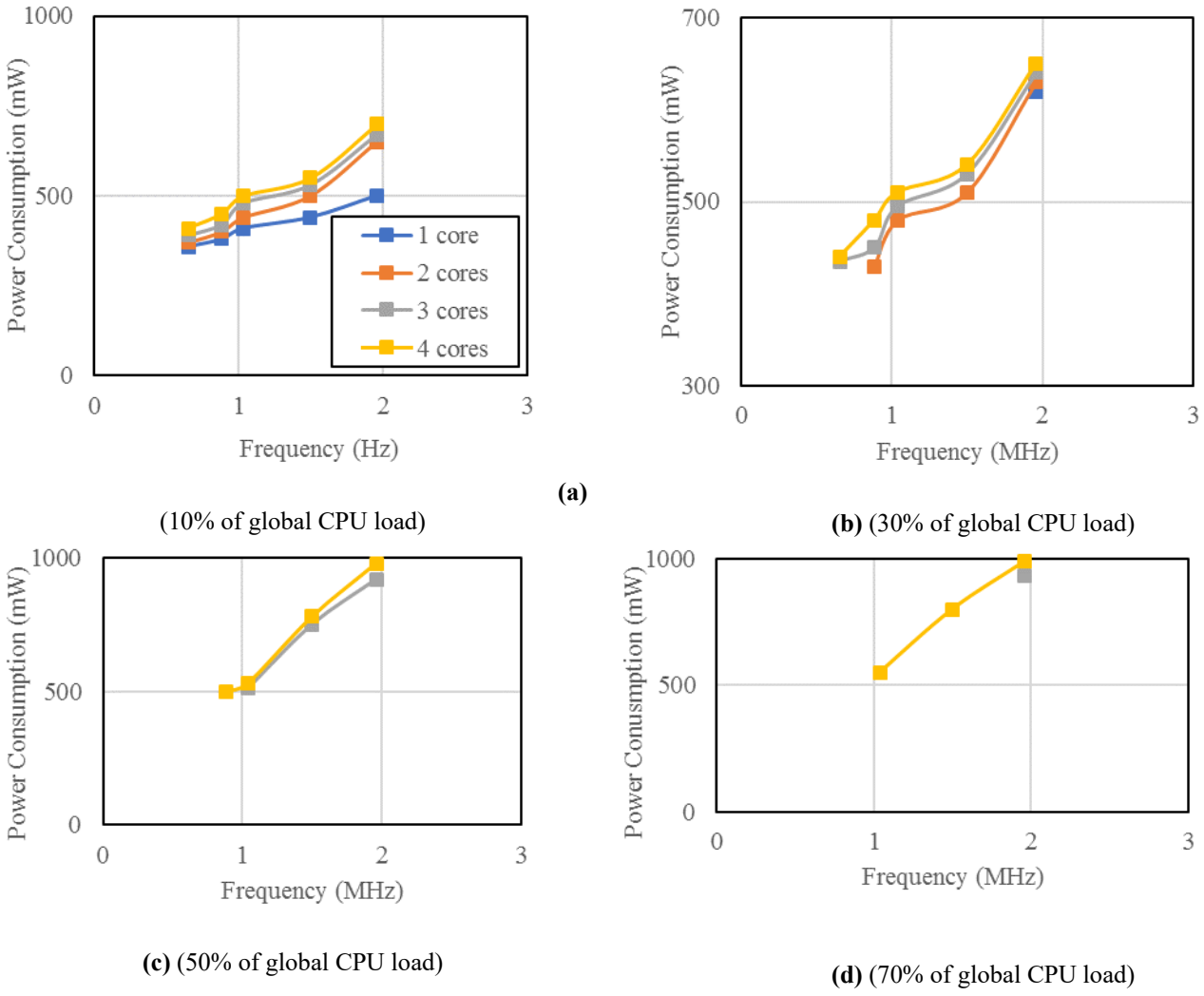
As we said before we are using a benchmark application which is characterized by a repetitive busy loop without memory accesses. The workload is executed during one minute under the minimal hardware resource features and then we tested all the different type of combinations which can provide the amount of workload asked. Furthermore, the default task scheduler does not have a major effect as mentioned in the constraints part above.

The results of our benchmark application are shown in Figures 5 (a), (b), (c) and (d). At a fixed frequency, using only one core (when the load is low enough), we can observe that this core is more efficient than 2, 3 or 4 cores. This behavior is repeated and increased along with the frequency. One explanation could be that when using one core, the three other cores are offline and that saves a lot of static power. Furthermore, at one point, the core 0 will enter an idle state which makes the overall computation power really low. That must significantly reduce the power consumption. We also observe that a minimal energy point is often achieved when more than the minimal number of cores is active. That allows the frequency of cores to be further reduced.

If we wanted to remove the period of idleness from our benchmark, it will appear that the ideal solution will be to maximize the number of online cores. Moreover, this will enable full throughput. That means that the use of little cores (and thus more of them) could improve the energy efficiency when correct operating points are selected. However, this will not be part of

this thesis because we will not focus on the difference that exists between big.LITTLE architecture (architecture embedding different type of cores with difference in capacity) but on the power efficiency and performance trade-offs of a simple multicore architecture (embedding same type of cores). Indeed, big.LITTLE architecture is bringing up new types of problems such how to schedule tasks between core, is a bigger core better to achieve this task along with the GPU … etc and asking for new types of solution [22].

We have evaluated all the type of combinations possible and their effectiveness achieving the hand-made benchmark. We also measured their power consumption. That allowed us to make up many optimal points for each specific workload and then, forming a curve. The idea will be that our design makes decision around that curve of optimal points.



(a) (10% of global CPU load)

(b) (30% of global CPU load)

(c) (50% of global CPU load)

(d) (70% of global CPU load)

Figure 5. Pow. Cons. over frequency when varying the operating point

3.5 BENCHMARKING PERFORMANCE

In this section, we want to understand how the default Android policy achieves performance requirement using a realistic benchmark. For that purpose, we chose the benchmark application GeekBench 4 [5]. This application performs a complex real-life benchmark on the available CPU

resources to push the limits of the system ensuring meaningful results by providing a value corresponding to the computing performance. The score represents the use of 1 single thread running on each of the active CPU cores. We first evaluated the influence of one single core.

Figure 6 confirms that the performance improves as the frequency increases, followed by an increase in power consumption. It seems that for a high frequency, say, 1.95GHz in our experiment, both the power consumption and the performance seem to reach a plateau. In summary, the CPU core has a stable behavior and stagnates toward the end because the gain induced by frequency increase does not help getting the workload done faster.

To see if such a behavior can be generalized when running multiple cores at the same time, we evaluated the ratio between performance and power consumption over the frequency range for one core and for four cores when running the benchmark application, as shown in Figure 7. We see that the ratio of one core is reasonably stable and increases slowly following a logarithmic trend. Theoretically, this is the best state we can reach. That proves our previous statement that one core is all the time giving good ratio power/performance. On the other hand, having 4 cores running at the same time shows a completely different behavior: After reaching a certain frequency (i.e., 960MHz), the ratio starts to decrease, which proves that the state of CPU cores is not ideal and the performance achieved is not worth the power consumption. The plot does not show any ideal state as there is no stable behavior (stagnation). That proves that involving too many cores at their highest computing state does not bring an ideal trade-off between power consumption and performance. Similar results can be obtained with fewer hardware resources in multicore case. Those results demonstrate that there is obviously a need to better balance the tradeoff between the power consumption and performance of the multicore

architecture in mobile devices. For a high computing demand, the Android default policy is not capable achieving a good trade-off.

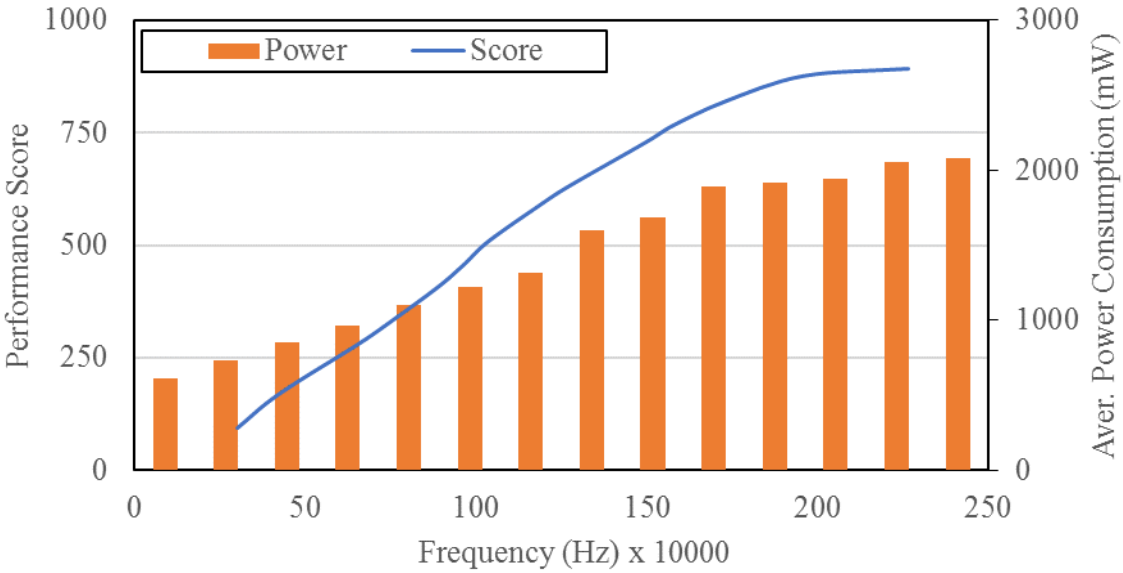


Figure 6. Pow. cons. and performance over freq. at 100% CPU utilization for 1 core

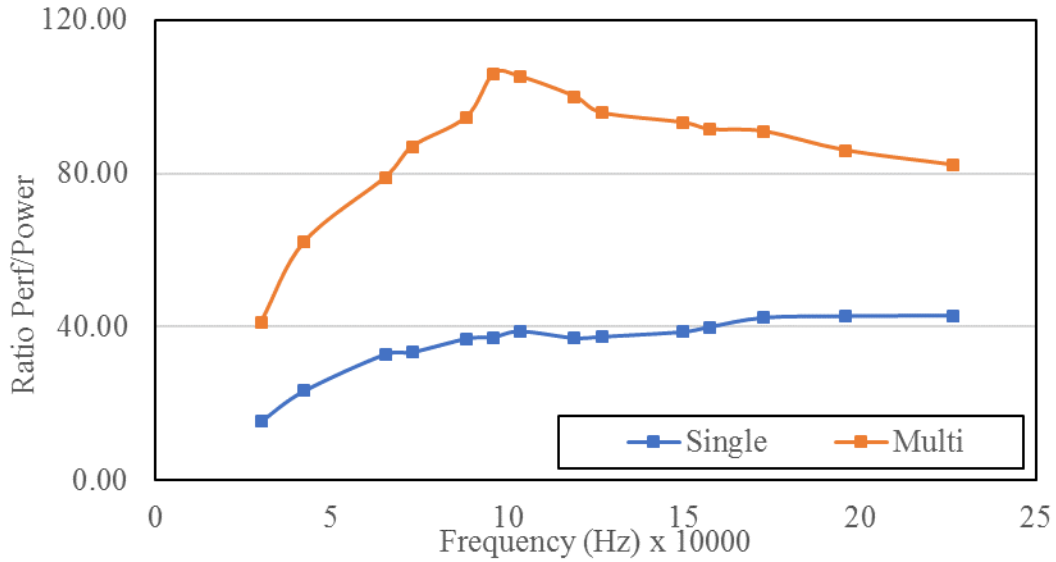


Figure 7. Performance/power ratio over CPU freq. for 1 and 4 cores

3.6 SUMMARY

This section has shown that power consumption and performance of a mobile phone are greatly impacted by the number of active CPU cores, core's frequency, and workload. The Android default policy gives good performance but power consumption raises a lot with it. That is why we tried to see how optimal points can be found. We changed the different features of cores (represents a percentage of hardware set) and measured how it was performing using a hand-made application which was loading the CPUs by busy loops. Now that we understood how we could adapt more smartly the hardware resources to the workload, we will see in the next section how we will design it. Note that the Android default policy does not give good results in terms of power consumption/performance ratio when handling multiple core architecture.

4.0 MODEL & DESIGN

In this section, we will present the theory behind the design of MobiCore. After that, we will describe the diagram flow of its algorithm and then introduce the implementation of our optimized solution. When developing our new model, we needed to setup some constraints. We first want to create power savings compared to the default policy in the Android system. Then we need to reproduce at least the same performance or, if possible, a better one. This method must be good in terms of time efficiency when switching to different states but still matching the requirements in performance and power consumption.

As we want to outperform the Android default policy, we will focus on high computing applications as it is on those one that the default policy is good in performance but lacks power reduction. We will represent those high computing applications by games.

4.1 CPU ENERGY MODEL

4.1.1 Basic Principles

The CPU power consumption model (P_{CPU}) is composed of the dynamic (P_d) and the static (P_s) power consumptions [6]. The former represents the power consumption when the core is busy and is approximately proportional to the frequency (f) and the square of the CPU voltage (V)

whereas the latter represents the power consumed when the core is in an idle state (this state does not depend on the workload as it is online but not busy; it only depends on the voltage):

$$P_{CPU} = P_d + P_s, \tag{1}$$

$$P_{CPU} = CV^2 f + V.I_{leak}, \tag{2}$$

where C is the capacitance dependent on the technology and I_{leak} is the leakage current. There is so little power consumption going from idle to active state that we won't count it in our model. We also assumed that going from online to busy does consume a little bit of power overhead which is counted in our static power variable. For n CPU cores, the total power consumption P_{total} can be calculated as:

$$P_{total} = n.P_{CPU} + P_{cache}, \tag{3}$$

$$P_{total} = n.(C.V^2.f + V.I_{leak}) + P_{cache}, \tag{4}$$

where P_{cache} is the power consumption due to accessing memory and independent on the number of cores. P_{cache} and V are dependent on the frequency. To get the energy consumption, we integrate the power consumption over the period of time as below:

$$E_{CPU}(\Delta t) = \int_0^{\Delta t} P(t).dt, \tag{5}$$

$$E_{CPU}(\Delta t) = T.P_{CPU}, \tag{6}$$

$$E_{total}(\Delta t) = T.n.P_{CPU} + P_{cache}. \tag{7}$$

Eq. (7) represents the energy consumption of n cores undergoing a global DVFS (same frequency for each core) during a period of time T.

MobiCore is looking for the best combination between the number of active CPU cores and their individual per-core frequencies in order to adapt better to the workload. As we build MobiCore upon the default governor, we re-evaluate the frequency from the previous choice made by the *ondemand* governor as:

$$n.f_{new} = n_{max}.f_{ondemand}.K, \quad (9)$$

where K is the current overall utilization of the phone, n is the number of active CPU cores, n_{max} is the maximum number of cores (here 4), f_{new} is the new frequency which will be calculated and $f_{ondemand}$ is the frequency which has been chosen by the *ondemand* governor. If we combine Eq. (2) and (9), we can estimate the power consumed by one CPU core with MobiCore as:

$$P_{CPU\_MobiCore} = \frac{n_{max}.f_{ondemand}.K}{n}.C.V^2 + V.I_{leak} \quad (10)$$

In our new methodology, we want the scheme to adapt quickly and more responsively to the workload. When facing a burst mode, our new design must be able to give correctly the amount of hardware to achieve the computing need. But we also want our new method to create more power savings when facing a slow mode. We can do that if we add a scaling value to our equation. In the Linux architecture, there exists a value which stands for the global CPU bandwidth. This value can be reduced or expanded by applying a small scaling factor (q) called quota. This scaling factor will affect the global utilization that will be put through the CPU cores, here, $K = K*q$. The value of the quota q depends on the level and variation of workload.

4.1.2 Validation for choosing off-lining

According to results in the DVFS literature [25], choosing a wrong operating point or believing in the knowledge that fewer active cores results in less power consumption can be catastrophic. Indeed, from previous experiments, it has been shown that power reduction is better when setting up the lowest possible frequency, but at a certain point putting more cores online is essential to offset the lost in capacity. In other words, having more online cores allows workloads to be processed at a more energy-efficient frequency with the same throughput and that could reduce power consumption. This could be true if the static power of our platform was low. We ran an experiment to measure the P_{static} for the maximum and the minimum frequency of one core of our platform and we found this: 120mW per core for f_{max}, and 47mW for f_{min}. That proves that our platform (Nexus 5) does not have a very low per-core static power. This proves that maximizing the number of online cores is not the optimal solution to save power and that the idle state does not imply important power reduction, thus race-to-idle concept is not a concept power-effective enough and won't be used in our new methodology.

These huge values in power consumption for an idle state can be explained by the fact that each core in the Nexus 5 is powered with an independent supply (which allows per-core DVFS). Idling cores in that configuration brings more power leakage as each core is a source of leakage. However, if we consider a platform where all cores are connected to the same voltage supply, there is fewer sources of power leakage as there is one voltage source. But that configuration does not allow per-core DVFS. Then, as idling cores won't bring enough power reduction, we will need to off-line cores.

A trade-off needs to be found between: the high dynamic power consumed by a higher frequency and the power reduction due to off-lining the core sooner as the workload has been

processed faster. Besides this trade-off, we noticed from our set of experiments that if the minimum frequency is not chosen, there is not enough power reduction. In the literature, solving such an issue calls for a dynamic power model.

Notice that if workloads are spinning without implying any period of idleness, it is more efficient to have more cores online as having a bigger throughput reduces execution time as well as static energy. This corresponds to a high dynamic workload and, as stated before, to address dynamic power with DVFS, as it is linear in frequency, race-to-idle principle can't be an ideal decision. This is another reason why our model is not considering idling cores but off-lining cores.

Finally, in order to get a more responsive policy to variation in workload, we are considering the size of the CPU bandwidth, which can be reduced or expanded a little bit. That can save more power when added to the previous analysis.

4.2 MODEL VALIDATION TO FIND OPTIMAL OPERATING POINTS

If we try to minimize the Eq. (10) to minimize the power consumption of each core, we will derive the equation according to the frequency. P_{static} and V are functions of *f*. Others variables are either constant or dependent on the workload. Then, the frequency *f* which will minimize P_{CPU} will be a function of those remaining values. We decided to give a constant value to C_{eff} too. It should be a function of instructions per cycle (IPC) as stated in [28] and [29]. IPC is one aspect of processor's performance. But as we are considering an optimal solution giving an acceptable performance, we will not consider extreme IPC values and this one is set to 0. Indeed, we are not looking at the best performance.

So now given that the workload is only characterized by its utilization *K*, we can predict the frequency which will minimize the per-core power consumption while achieving the required workload. The logic is then the same as the one described in Section 3.4. For a certain workload, many combinations of hardware resources are possible. The best one is chosen by our model.

Increasing the frequency represents the choice at low workload whereas increasing the number of active core appears at higher utilization. The curve which represents the combination looks like the scar on Harry Potter's face.

- 1 core is online and we increase its frequency until the point where 2 cores at their minimum frequency give the same amount of workload;

- then we vary the frequency of the two cores and then we reach the point when 3 cores at their minimum frequency gives the same amount of workload;

- we vary their frequency; and finally, we reach the point where 4 cores at their minimum frequency can process the same amount of workload;

- so we switch and then we increase their frequency.

But for our new design, we are also considering the power consumption of each combination. The system will then simply choose which combination gives the best amount of workload for the least amount of power.

4.3 SUMMARY

In this section, we have presented that running a workload on more cores may improve power reduction. Either by choosing a more energy efficient running frequency or by reducing the execution time. We have presented a new dynamic model which considers minimizing the per-

core static power consumption by off-lining cores. We observed a high per-core static power consumption on the platform we are using due to the fact that we are able to process a per-core DVFS. That proves that off-lining cores will help in making important power reduction. But also tells us that the old basic principle working great on previous platforms with global DVFS (race-to-idle principle) won't give an optimal solution. This calls for creating a new method which predicts an optimal operating point considering the sources of potential power reduction. We validated our simple model (which implies basic assumptions) by minimizing the equation. This model makes the best choice for the amount of workload which has to be processed and the least power consumption.

5.0 MOBICORE DESIGN AND IMPLEMENTATION

In this section, we are introducing the new method called MobiCore which is based on the previous analysis. The design and the flow chart of this new policy will be described as well as how we implemented it on our mobile platform.

5.1 PERFORMANCE CHARACTERIZATION

The performance of MobiCore is measured in frames per second (FPS), which reflects the execution time of the process in terms of frames per second. If the frequency at which the process is running is high, the FPS will be high as the execution time per frame will be shorter. But graphics need to be computed as well as many other things. We set the GPU frequency at its highest so we assume that there is no bottleneck coming from it. Only the allocated CPU resources will affect the rendering. Moreover, it is not necessary to achieve an as-high-as-possible FPS. Usually the required FPS is fixed, e.g., 60 for games and movies. To see whether this value was reached by the default policy in Android, we performed a series of measurements on several games on the Nexus 5. Our results show that most of the games were running between 15 and 20 FPS though the gaming experience was unaffected by this value (means smooth rendering). Therefore, we consider that this FPS range is acceptable for our experiments.

5.2 DESIGN OF MOBICORE

As stated before, the equation that is using MobiCore implies to re-calculate the frequency of each core. Prior to these calculations, we will determine if we can reduce/expand the allocated CPU bandwidth based on some workload analysis. MobiCore is an adaptive scheme in the sense that it must respond to variations in workload so that it can answer the dynamicity of some applications. For that purpose, in the Android architecture, there exists a value called quota which can be modified if a scaling factor is applied. This is a global value which affects all the CPU cores so that any modifications must be carefully evaluated. MobiCore analyzes the variation in global utilization between time step t and time step t-1. If the difference is above a certain threshold and positive, we are facing a burst mode; if it is negative, or say, the computing need is suddenly low, we are facing a slow-mode. For those two modes, we respectively allocate the entire bandwidth or apply a small scaling factor to reduce the allocated bandwidth. An example of the reduction of bandwidth is shown in Table 2 (Algorithm 4.1.2). Note that we analyze the variation in workload only if the overall load is below a certain threshold; if the overall workload is high at t and t-1, variation will be inexistent but CPUs will still need a high bandwidth.

After that analysis, we re-evaluate the number of active CPU cores; we based our reasoning on a simple threshold: if the individual workload of a core is under 10%, we assume that we can turn it off.

Finally, based on this new information we calculate the new per-core frequency. The implementation is shown and explained in the next subsection.

Table 2: Example of the C code of MobiCore

Algorithm 4.1.2 Bandwidth Reduction

Input: utilization, quota, scaling_factor
Output: quota

```
1:  for each sampling period
2:  quota = utilization
3:      if utilization(t) <40
4:          if Δ utilization (t – t-1) < downThreshold
5:              scaling_factor = 0.9
6:              quota = quota*scaling_factor
7:          endif
8:          if Δ utilization (t – t-1) > upThreshold
9:              scaling_factor = 1
10:             quota = quota*scaling_factor
11:         endif
12:     endif
13: end for
```

5.3 LOW CHART & IMPLEMENTATION

Figure 8 illustrates the diagram flow of MobiCore. First, the on-demand governor is run and the default DVFS is applied. Then, it is time to reduce or expand the CPU bandwidth: in other words, we check if the CPU load is high or low and then if there is an obvious high variation of load between time step t and t-1. Depending on the results, it will be decided whether the quota will be scaled down or not. After that, MobiCore re-estimates the number of the required active CPU cores based on the percentage of utilization attributed to each of those cores. A new frequency for each core is calculated based on the one previously calculated by the default

Android system but also based on the new number of active CPU cores, the per-core utilization and the bandwidth allocated to the cores.

Note that we do not have to setup a new frequency threshold as looking for a good operating point will automatically switch to add a new core instead of raising the frequency too high.

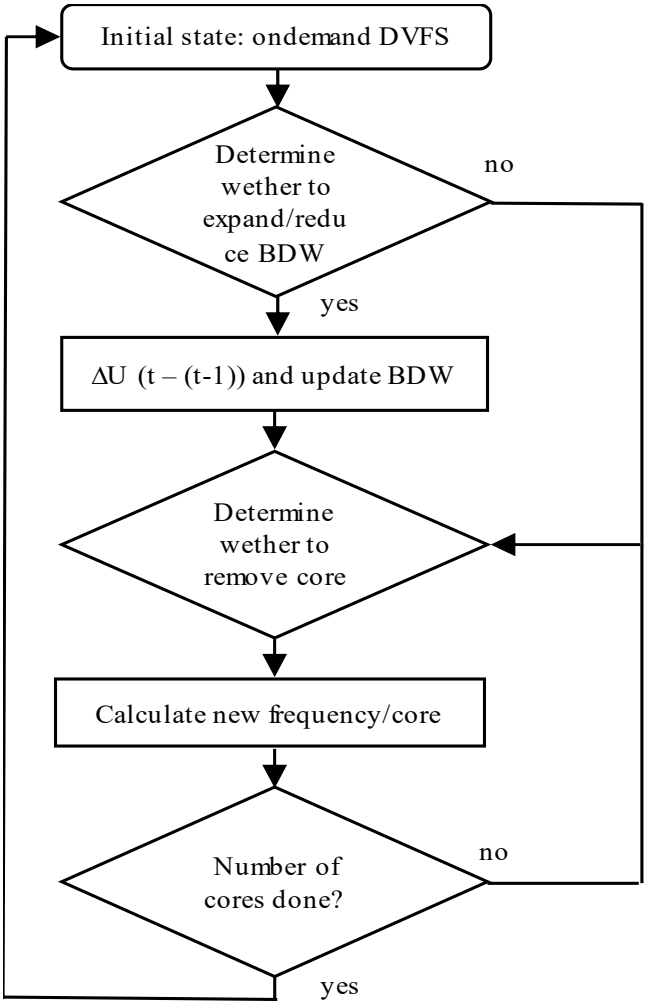


Figure 8. System Diagram Flow of MobiCore

We implemented MobiCore on a Nexus 5 mobile device. All CPU features that are tweaked are easily accessible and modifiable in the Android Linux architecture, which is an open

source project. MobiCore is implemented as a CPU governor (at the *userspace* governor location as stated before) in the Linux architecture. It is written in C and sent to the system by command line through *adb shell*. As stated before, it is based on the existing *ondemand* governor and integrating some features of the *hot-plug* policy as well as having a control over the CPU bandwidth.

5.4 SUMMARY

This section presented the whole flow chart of MobiCore as well as how it has been implemented and on which platform. One example of the pseudo code has been shown. We described in details the flow-chart of MobiCore, which components are affected and which steps are happening at what time. We will now begin the tests and experiments and compare this whole new design called MobiCore to the Android default policy.

6.0 EVALUATION: THE MOBICORE EXPERIMENT

In this section, we will present the results and comparisons collected from video games and from some benchmarks (hand-made skeleton application and GeekBench 4) on our Nexus 5 platform. Total of 5 modern representative games are tested, including Real Racing 3, Subway Surf, Badland, Angry Birds, and Asphalt 8 (numbered from 1 to 5). Every gaming session lasts 2 minutes and is played under both MobiCore and the Android default policy. The games have been designed to run on multicore architecture and are multithreaded. Power measurements are performed using the power Monsoon tool and the kernel application is run in the background to get the values of the CPU features (frequency, utilization, time, etc.). Running this kernel application does not involve visible computation. Comparisons are made between MobiCore and the Android default policy. The phones' airplane mode is set on to avoid any power consumption from undesirable communications.

6.1 EFFECTIVENESS IN POWER SAVINGS

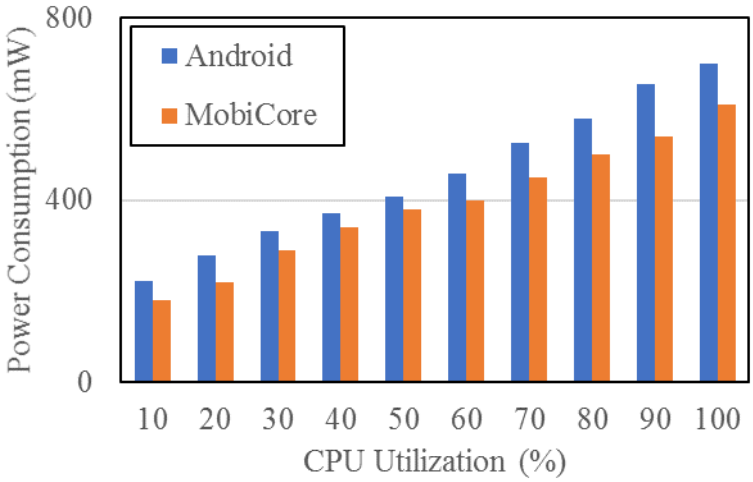
6.1.1 Basic benchmarks

In this sub-section, we are evaluating the behavior of MobiCore on different types of benchmarks: one quite static thanks to our basic hand-written application and another one more dynamic thanks to GeekBench 4.

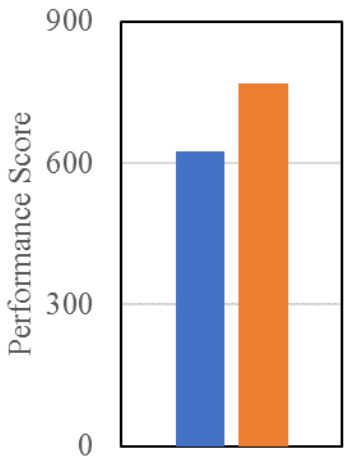
First, we evaluated MobiCore with our basic skeleton application which runs busy loops on the processors. We varied the workload from 10 to 100% as before and compared with the default Android policy. The workload is really stable so the results obtained will represent well enough the real effective performance of MobiCore compared to the Android default policy. In a second part, we tested MobiCore on the Geekbench 4 benchmark. The score will give us an idea on how MobiCore behaves in a more dynamic environment and how much it will outperform the Android default policy.

From Figure 9(a), we can see that MobiCore always achieves power reduction when compare to the Android default policy for each different workload for the hand-written benchmark. The worst-case of power differences is 6.8% for 50% of CPU workload whereas the best is 20.9% for 20% of CPU workload. MobiCore on average achieves 13.9% of power reduction compared to the Android default policy for this static benchmark.

The results of the Geekbench 4 benchmark are shown in Figure 9(b). It appears that MobiCore is well-adapted to a dynamic environment. According to this, MobiCore outperforms the Android default policy by almost 23%.



(a) Power consumption on hand-written benchmark



(b) Score on GeekBench 4

Figure 9: Performance and power consumption on 2 benchmarks

These measurements made with these two benchmarks show that MobiCore always achieves power reduction for dynamic and more static workload. This proves the efficiency of the original design of MobiCore. We will pursue experiments and tests on more highly dynamic applications in the next sub-section to go further in the exploitation of MobiCore.

6.1.2 Representative games

In this sub-section, we are testing our method on five heavy computing applications represented by games described at the beginning of the section. Figure 10 presents the total average power consumption of the phone during each gaming session with the Android default policy and with MobiCore.

We can see that running games under MobiCore is generally consuming less power compared to running the same games under default Android policy. Power savings of each game

varies from 0.04% (for Real Racing 3) to 11.7% (for Subway Surf). On average, Mobicore achieves 5.3% of power savings compared to using Android default policy.

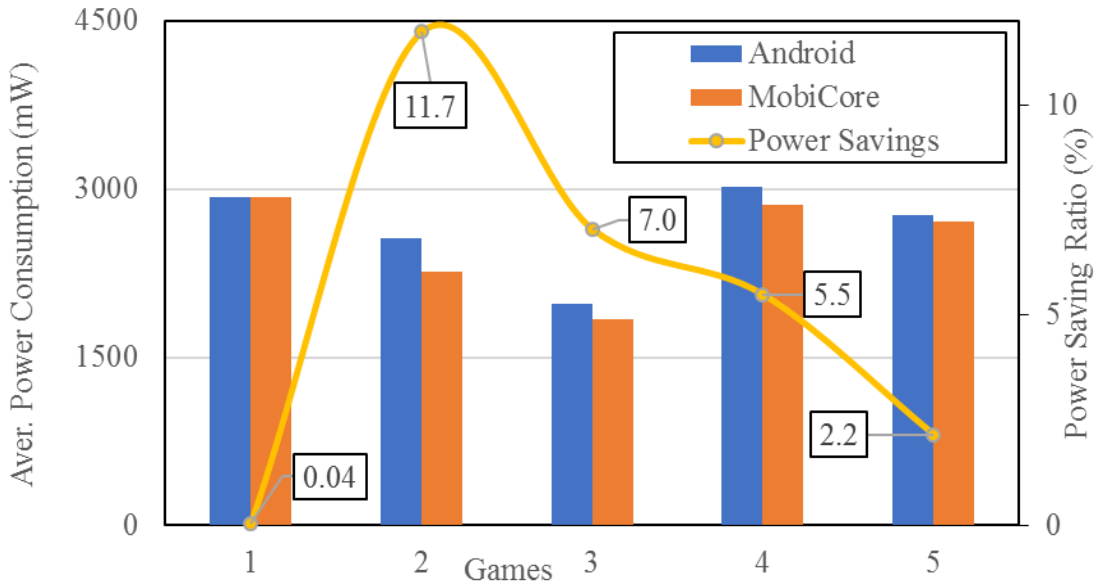


Figure 10. Average power consumption comparison

We can conclude that our new scheme does bring power reduction for heavy computational applications. Substantial power reductions are not created all the time and this must be due to some specific dynamicity of games. If no power reductions are made, we can at least say that MobiCore achieved the same power consumption than the Android default policy.

6.2 EFFECTIVENESS IN PERFORMANCE

As stated before, we are measuring the performance reached by the average FPS value reached when running games. Our measurements in Figure 11 show that the Android default policy always manages to achieve a higher FPS than MobiCore for each game. But the FPS value with

MobiCore remains in the range between 15 and 20, which is generally acceptable for running a game as we discussed in Section 5.1. Although we did not perform a detailed human visual perception test, we may still conclude that MobiCore is giving a good level of system performance while acknowledging that Android default policy is overdoing that. When looking at the FPS ratio, we can say that, on average, MobiCore achieves 22% fewer FPS than the Android default policy does.

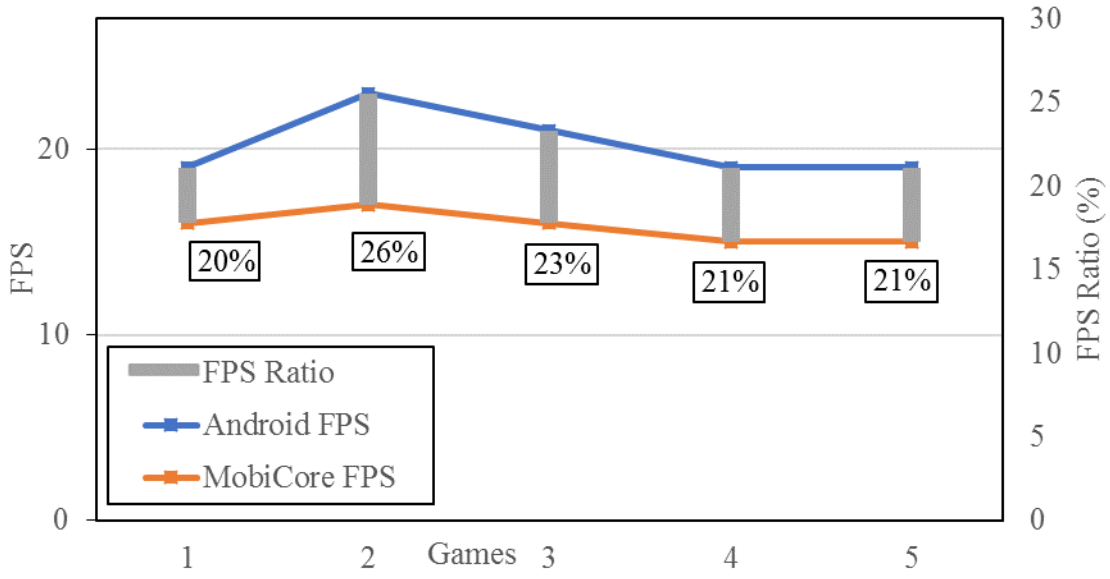


Figure 11. Average FPS reached and FPS ratio

That means that MobiCore is a good fit for gaming session in terms of rendering. That gives us one more good reason to say that MobiCore might be better than the Android default policy.

6.3 EFFECTIVENESS IN USING HARDWARE COMPONENTS

In this subsection, we will measure how much the hardware components are stressed with MobiCore and compare it to the Android default policy. We performed some experiments to compare the average frequency of the cores, the average global utilization of the cores, and the average number of the active CPU cores adopted by both policies during each gaming session.

Figure 12 shows the variations of the average frequency of the cores under MobiCore and under the Android default policy and the average number of active CPU cores during each gaming session. MobiCore generally requires a lower average frequency of the cores in these games except for Real Racing 3, where the average frequency of the cores is 0.5% higher than that of Android default policy. It is because a fixed number of the active cores is sufficient to handle the dynamic computation need of the games, leaving no room for MobiCore to further optimize. Nonetheless, in average, MobiCore achieves 22.5% lower frequency of the cores. The average number of active CPU cores under the Android default policy is always higher than the one under MobiCore: on average, MobiCore uses 2.52 CPU cores when Android uses 2.75.

First, that proves us that MobiCore does use lower frequency but also surprisingly that it uses fewer CPU cores. That could be explained by the fact that we are modifying the global CPU bandwidth and that might affect the choice on the number of cores.

Figure 13(a) shows the average CPU load of both policies for different games. The cores under the Android default policy is on average 3.1% busier than that under MobiCore and a positive workload reduction is observed at all games. That is shown more clearly in Figure 13(b).

After observation, we can draw some conclusions that correlate with our previous statements. The power saving is mainly coming from DVFS than DCS. Indeed, our highest power saving (11.7%) corresponds to the largest difference in average frequency of the cores

(43%) and the highest number of cores usage (3.9) at Subway Surf. But because games mainly remain on the effectiveness of the code, we can't generalize this statement. Indeed, as we saw, on the other hand, Real Racing 3 uses only 2.2 cores on average with even negative frequency reduction, leading to a very marginal power saving (i.e., 0.03%).

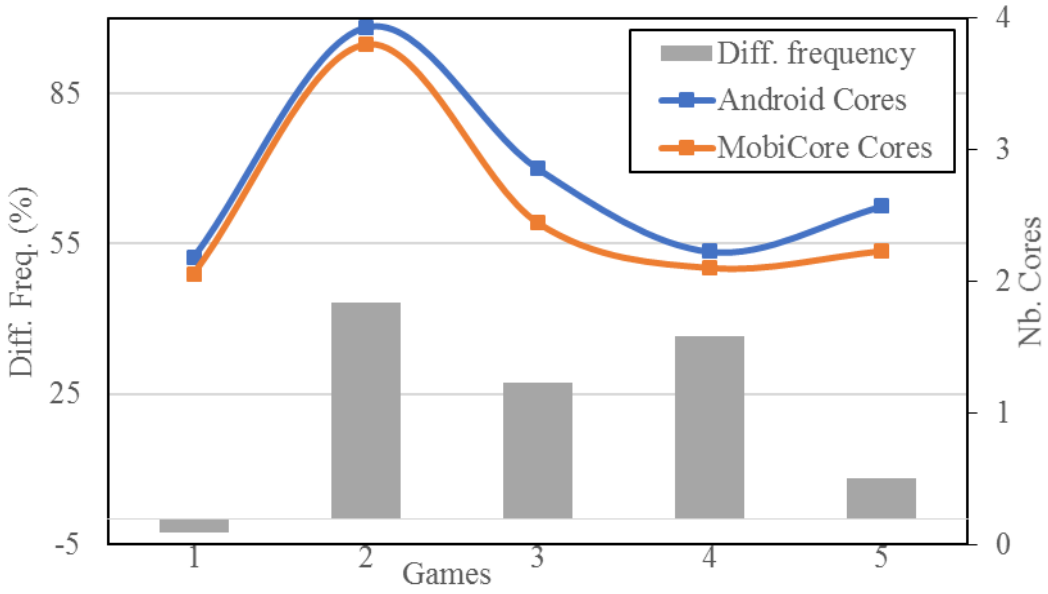
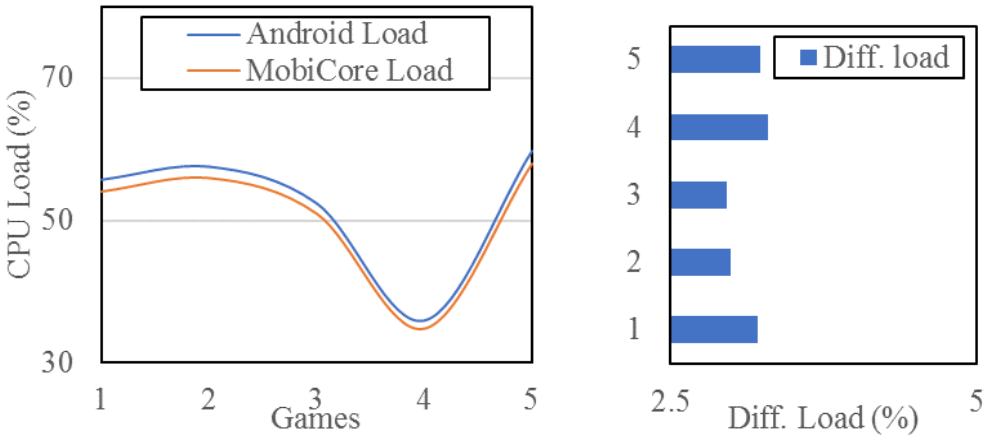


Figure 12. Average frequency difference and number of cores



(a) Average load (b) Load variation

Figure 13. CPU load Stress Level

6.4 COMMENTARIES

Theoretically, it seems that the main limitation of our work is the assumptions we made before for designing our model.

Our simple assumptions can certainly not be generalized due to the wide variety of type of processors but also by the fact that a processor works in relation with many other components. We tried to minimize their effect but we can't say that they don't have any. Thus, the wide variety of combinations of hardware components is also a major obstacle to our design. MobiCore is then unlikely to apply to all types of system-on-chip.

It seems that MobiCore is adapted for dynamic and less-dynamic workload as the hand-made and GeekBench 4 benchmarks both gave good results (i.e. 14% and 23% power savings, respectively). More dynamic applications are bringing less power reduction (i.e. 5.3% in average and up to 11.7%). This makes sense as the algorithm is sensitive to changes in workload. Also, they are more subject to badly written code which means that the games do not take into account the whole capacity of modern multicore architecture.

6.5 SUMMARY

As a summary, we can state that MobiCore achieves good efficiency when choosing between the number of cores and the average frequency. It does choose the best working frequency adapted

to the workload as we are able to make power savings mainly due to DVFS and still maintain good performance.

We note that the slight degradation in performance (reflected as fewer FPS) when using MobiCore is because we are using fewerhardware resources. The tradeoff between the performance degradation and power saving is worth a further investigation. It is essential to mention that MobiCore is a better choice than Android for heavy workload. The default Android policy will quickly adapt the CPU resources to the demand either raising up the frequency or adding more cores leading to a good performance but important power consumption. Whereas MobiCore is looking to find a better combination of the CPU resources to make more power reduction while being more responsive to workload as it is adapting the CPU bandwidth along with it. This scheme uses fewer hardware resources and as seen in Figure 10 with Subway Surf (which is using lots of resources with Android) more power savings are made.

7.0 CONCLUSION & FUTURE WORK

In this paper, we proposed MobiCore, a workload-based adaptive hybrid CPU aware policy for mobile devices to achieve power savings compared to the Android default policy. MobiCore finds, in one step, the ideal number of active CPU cores as well as each of their frequency and controls the allocated bandwidth to reach the best CPU operating state in terms of performance and power savings tradeoff. The power consumption analysis is provided on a Nexus 5 running under Android 6.0. MobiCore has been tested on several heavy computational applications represented by games. According to our experiments, MobiCore is able to make substantial power savings of up to 11.7% compared to the default policy in Android when playing games.

Future research topics could be exploring more affine techniques combining the characteristics of every component in a mobile device. As components in a mobile device are sharing the same platform, a sort of global DVFS policy could be applied considering the effect of each component as well their own bottleneck to better allocate the resources according to the workload. This could help find the best overall state for phone. Taking into consideration each component will make sure none of them is reaching a bottleneck state and slowing down the processing time or performance of the mobile device.

BIBLIOGRAPHY

[1] Flir Official Website. Available: http://www.flir.com/security/display/?id=56766

[2] Li, S., and Mishra, S., “Optimizing power consumption in multicore smartphones,” in IEEE Trans on Parallel and Distributed Computing, Sept. 2016, pp. 124-137.

[3] Gao, C., Gutierrez, A., Rajan, M., Dreslinski, R. G., Mudge, T. and Wu, C. J.,“A Study of Mobile Device Utilization,” in IEEE International Symposium on Performance Analysis of Systems and Software, Mar. 2015, pp. 225-234.

[4] Nexus 5 Specifications. Available: http://www.gsmarena.com/lg_nexus_5-5705.php

[5] Google Play Store, Geekbench 4. Available: https://play.google.com/store/apps?hl=en

[6] Kim, M., Ju, Y., Chae, J. and Park, M., "A Simple Model for Estimating Power Consumption of a Multicore Server System," in International Journal of Multimedia and Ubiquitous Engineering, 2014, 9.2: pp.153-160

[7] Pallipadi, Venkatesh, and Starikovskiy, A., "The ondemand governor," in Proceedings of the Linux Symposium, July 2006, Vol. 2., pp. 215-230.

[8] Chandana, Hari, K. S., and Maliye, S., "Android activity based intelligent hotplug control," in Microelectronics, Computing and Communications (MicroCom), 2016 International Conference on., Jan. 2016, pp. 1-4.

[9] Kim, Min, J., Kim, M., and Chung, S. W., "Application-aware scaling governor for wearable devices," in Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on. IEEE, Sept. 2014, pp. 1-8.

[10] Lin, C.C., Chang, C.J., Syu, Y.C., Wu, J.J., Liu, P., Cheng, P.W. and Hsu, W.T., "An Energy-Efficient Task Scheduler for Multi-core Platforms with Per-core DVFS Based on Task Characteristics," in 2014 43rd International Conference on Parallel Processing. IEEE, Sept. 2014, pp. 381-390.

[11] Dai, J., Liu, W., Lin, X., Ye, Y., Xiao, C., Wu, K., Zhuge, Q. and Sha, E.H.M., "User Experience Enhanced Task Scheduling and Processor Frequency Scaling for Energy-Sensitive Mobile Devices," in High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security

(CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, Aug. 2015, pp. 941-944.

[12] Tseng, P. H., Hsiu, P. C., Pan, C.C. and Kuo, T,W., "User-centric energy-efficient scheduling on multi-core mobile devices," in Proceedings of the 51st Annual Design Automation Conference. ACM, Jun. 2014, pp. 1-6.

[13] Kim, S., Kim, H., Kim, J., Lee, J., and Seo, E., "Empirical analysis of power management schemes for multi-core smartphones," in Proc. of the 7th International Conference on Ubiquitous Information Management and Communication. ACM, Jan. 2013, p. 109.

[14] Das, Anup, et al., "Hardware-Software Interaction for Run-time Power Optimization: A Case Study of Embedded Linux on Multicore Smartphones," in Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on, Jul. 2015, pp. 165-170.

[15] Halpern, M., Zhu, Y. and Reddi, V.J., "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction," in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Mar. 2016, pp. 64-76.

[16] Monsoon power. Available: http://www.msoon.com/LabEquipment/PowerMonitor/

[17] Deca-core phone. Available: https://techcrunch.com/2016/08/25/xiaomi-note/

[18] W. Lloyd Bircher and Lizy K. John., "Analysis of dynamic power management on multi-core processors", in SC'08, Kos, Greece, Jun 2008.

[19] Etienne Le Sueur and Gernot Heiser, "Slow down or sleep, that is the question", in 2011 USENIX ATC, Portland, OR, USA, Jun 2011.

[20] W. Lloyd Bircher and Lizy K. John, "Analysis of dynamic power management on multi-core processors", in SC'08, Kos, Greece, Jun 2008.

[21] M. Ghasemazar, E. Pakbaznia, and M. Pedram, "Minimizing energy consumption of a chip multiprocessor through simultaneous core consolidation and DVFS", in ISCAS, pages 49–52. IEEE, 2010.

[22] Peter Greenhalgh. "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7", in ARM White Paper, 2011.

[23] Linux architecture documentation, Linux CPUFreq Governors. Available: https://android.googlesource.com/kernel/common/+/a7827a2a60218b25f222b54f77ed38f57aebe08b/Documentation/cpu-freq/governors.txt

[24] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. "Critical power slope: understanding the runtime effects of frequency scaling", in 16th Int. Conf. Supercomp., pages 35–44, New York, NY, USA, Jun 2002.

[25] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser "Koala: A platform for OS-level power management", in 4th EuroSys, Nuremberg, Germany, Apr 2009.

[26] LG Google Nexus 5. Device Specifications. Available. http://www.devicespecifications.com/en/model/3dfd2923

[27] Linux Hotplug policy. Available. https://www.kernel.org/doc/local/hotplug-history.html

[28] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, and Karsten Schwan, "The forgotten 'uncore': On the energy-efficiency of heterogeneous cores", in 2012 USENIX ATC, Boston, MA, USA, Jun 2012.

[29] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS", in IGCC, pages 1–8. IEEE, Jul 2011.