Power and Energy Analysis on Odroid-XU+E and Adaptive Power Model

In Hwan Baek
University of California Los Angeles
chris.inhwan.baek@gmail.com

Xiangrui Liu
University of California Los Angeles
xrliu@ucla.edu

ABSTRACT

Today, the increasing demand for more powerful processors on mobile devices has made the power dissipation increasingly important to consider. To assist the developers to make design decisions for better power efficiency, researchers have proposed several power models. However, the conventional power model generation process is done with power measurement at the device-level and is very time consuming. To address the problem regarding the device-level power measurement, we propose a CPU-level power model that is generated with the power sensor data on each core of Exynos 5 Octa 5410. To further improve the accuracy of our power model and overcome time consuming nature of model training, we propose a fast adaptive power model, which automatically adjusts parameters in the model for different types of tasks. The demonstration of the adaptive power model is given with an energy consumption prediction program, which also investigates execution time models and energy consumption with respect to the CPU frequency. To evaluate our work, we calculated power consumption with the adaptive power model and power consumption with an offline trained model. The result shows that adaptive power model after self-adjustment fits to the offline trained model with a small error.

Categories and Subject Descriptors

I.6.0 [**Computer Methodologies**]: Simulation and Modelinggeneral

General Terms

Power, energy

Keywords

Power model, execution time model, measurement

1. INTRODUCTION

Today, the increasing demand for more powerful processors on mobile devices has made the power dissipation increasingly important to consider. However, many software developers lack knowledge on the power consumption behaviors of devices. As a consequence, there are many unnecessarily power-hungry applications for mobile devices [1].

To assist the developers to make design decisions for better power efficiency, researchers have proposed several power models. A small selection of devices are investigated and the power models are derived with power meters attached to the selected devices. There are two main drawbacks for such power model derivation process. First, the power models are generated at the device level, which would lead to inaccuracy in the generated power models for each components. To measure power dissipation on a specific component, a common approach is turning off other components. For instance, cellular, WiFi, GPS, etc. are turned off while measuring the power dissipation on CPU. The problem is that not all components, such as RAM, can be turned off. The power dissipation on such components and even leakage power on the whole device may have a negative impact on power measurement accuracy. Second, the power models may be accurate for the specific devices, from which the models are derived, and inaccurate for other devices. The problem is that it is nearly impossible to generate power models for all available devices with such conventional approach because it is very time consuming and painstaking to generate power model for the wide range of mobile devices available today.

ODROID-XU+E and ODROID-XU3 are Linux/Android platforms that feature Samsung Exynos processors, the same processors installed in Samsung Galaxy S4 and Galaxy S5. These platforms are great for characterizing these processors. These processors have ARM big.LITTLE architecture. This architecture is discussed in section 2.2. The most interesting feature is that these platforms have on-board power sensors for the big CPU cores, the little CPU cores, the GPU, and the RAM. Thus, they allow component-level power measurement. The inaccuracy issue with device-level measurement can be overcome with these sensors.

With the capability of component-level power measurement, we propose a new approach to generate power model for CPU. Using collected power sensor reading data on the big CPU core and the little CPU core, we trained a power model. We have studied of the variation of power consumption on different types of tasks. To further improve the accuracy of our power model, we propose an adaptive power model, which automatically adjusts parameters in the model for different types of tasks.

2. BACKGROUND

In this section, we discuss about the platform, on which we implemented our system framework. The operation modes of big.LITTLE architecture are described and the core switching method in CPU migration mode is explained. Dynamic voltage frequency scaling (DVFS) is discussed because our models are at CPU-level and the power dissipation on CPUs largely depends on the supply voltage and the frequency.

2.1 ODROID-XU+E

Odroid-XU+E is a new generation of computing device with more powerful, more energy-efficient hardware and smaller form factor. Offering open source support, the board can run various flavors of Linux, including Ubuntu and the Android 4.2. By adopting e-MMC 4.5 and USB 3.0 interface, the ODROID-XU boasts high speed fast data transfer, a feature that is increasingly required to support advanced processing power on ARM devices. The board is equipped with four big cores (ARM Cortex-A15 up to 1.6GHz) and four small cores (ARM Cortex-A7 up to 1.2GHz).

2.1.1 Power monitor

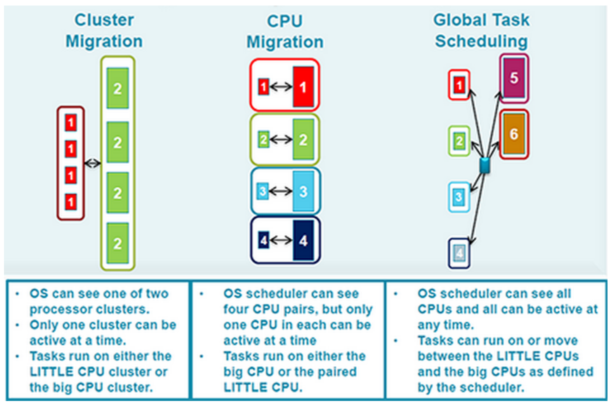
ODROID-XU+E has an integrated power analysis tool. This package contains a special ODROID-XU board which has four current/voltage sensors to measure the power consumption of the Big A15 cores, Little A7 cores, GPUs and DRAMs individually. The professional developers can monitor CPU, GPU and DRAM power consumption via included on-board power measurement circuit. By using the integrated power analysis tool, developers will reduce the need for repeated trials when debugging for power consumption and get the opportunity to enhance and optimize the performance of their CPU/GPU compute applications, and therefore keeping power consumption as low as possible.

2.2 big.LITTLE Architecture

By using the big.LITTLE heterogeneous computing architecture, the Samsung Exynos 5 Octa 5410 couples slower, low-power processor cores with relatively faster, high-power processor cores in order to reduce power consumption. The ”big” or faster cores are used for computation-intensive tasks such as gaming, whereas the ”little” or slower cores are used for less intensive tasks.

2.2.1 CPU Migration

The big.LITTLE architecture has three operation modes to utilize and arrange the high performance big CPU cores and power efficient little CPU cores. These three modes are cluster switching, in-kernel switcher (CPU migration), and heterogeneous multi-processing (Global task scheduling), which are compared in Figure 1 Exynos 5 Octa 5410 was the first architecture that featured cluster switching mode, the simplest mode of the three. In cluster switching mode, the big cluster consists of identical big CPU cores such as Cortex-A15 and the little cluster consists of identical little CPU cores such as Cortex-A7 [2]. In the perspective of the Operating System scheduler, only one cluster exists at a time. In other word, one cluster can be active at a time. Depending on the load on the CPU, all relevant data migrate to



Cluster Migration	CPU Migration	Global Task Scheduling
• OS can see one of two processor clusters. • Only one cluster can be active at a time. • Tasks run on either the LITTLE CPU cluster or the big CPU cluster.	• OS scheduler can see four CPU pairs, but only one CPU in each can be active at a time • Tasks run on either the big CPU or the paired LITTLE CPU.	• OS scheduler can see all CPUs and all can be active at any time. • Tasks can run on or move between the LITTLE CPUs and the big CPUs as defined by the scheduler.

Figure 1: ARM big.LITTLE operation mode (Source: ARM)

another cluster via L2 cache. One of the main drawbacks of this mode is that within one cluster, the load on each core will vary. In other words, one core may have heavy load while another core has little load.

CPU migration mode was implemented by Linaro to overcome such imbalanced CPU load. In CPU migration mode, one big CPU core and one little CPU core are paired up and the OS scheduler sees each pair instead of a cluster. The migration decision is taken at the cpufreq driver level. If the required frequency is too high or too low for the current CPU core, migration to another core is requested to the In-kernel Switcher logic. In order to approximate a linear scale across the combination of A7 (little core) and A15 (big core), the virtual frequency set for A7 is scaled down by half. In other words, A7 core’s maximum available frequency 1.2GHz is scaled down to 600MHz. If the requested frequency is greater than 600MHz, A15 is activated. If a frequency less than or equal to 600MHz is requested, A7 is activated. Based on the collected data on the power dissipation of A7 and A15 in different frequencies, we concluded that CPU migration mode was in use on the ODROID-XU+E.

2.3 Dynamic Voltage Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a commonly-used power-management technique where the clock frequency of a processor is decreased to allow a corresponding reduction in the supply voltage.

2.3.1 Idle states and Performance states

In order to effectively manage power consumption, modern processors have several idle states (C-states in ACPI standard) and processor performance states(P-states). Different levels of idle states have different power consumption savings and wakeup latency overheads [3]. In other words, the deeper a processor sleeps the less power is consumed but longer it takes to wake up. Performance states, on the other hand, are related to CPU frequency and voltage. A processor in a higher performance state runs at lower frequency and voltage. The control of the performance states is done by the CPUfreq subsystem.

2.3.2 CPUfreq subsystem

Governor	Summary
Interactive	Designed for latency-sensitive workload. Significantly more responsive than other governors.
Performance	Sets to the maximum frequency available.
Powersave	Sets to the minimum frequency available.
Userspace	Only user can set the frequency.
ondemand	Sets to the maximum frequency when CPU utilization exceeds threshold and then steps down to lower frequencies if the utilization is less than the threshold until the lowest frequency is set.
conservative	Gradually increases or decreases the frequency while checking the utilization.

Table 1: CPUfreq Governors

The CPUfreq subsystem provides users ability to tune the performance state management. There are six in-kernel governors for the CPUfreq subsystem: interactive, performance, powersave, userspace, ondemand, and conservative. Each has a different policy to select performance states as shown in Table 1.

3. SYSTEM FRAMEWORK

We first discuss the power consumption at CPU level. Based on this, we explain our approach to generate a CPU-level power model. The adaptive power model design incorporates the pre-trained power model. The design is explained in details. Execution time models are generated to analyze the performance and energy consumption with respect to the task load and the CPU frequency. The our approach to execution time model generation is described.

3.1 Power Model

In order to explore the factors contributing to the CPU power consumption, it is necessary to look at the CMOS integrated circuit power consumption equation [4]:

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage} \tag{1}$$

The equation consists of three components. Figure 2 illustrates this three components of transistor power consumption. The first component estimates the dynamic power consumed by charging and discharging the capacitance at each gates output. The second component estimates the power expended by the "short-circuit" current which momentarily flows between the supply voltage and ground when the CMOS gates switch. The third component estimates the power due to current leakage.

In practice, we observe that the dynamic power component dominates the total CPU power consumption. The dynamic power consumed by a CPU is approximately proportional to CPU frequency and the square of the CPU voltage [5]:

$$P_{dynamic} = \alpha C V^2 f \tag{2}$$

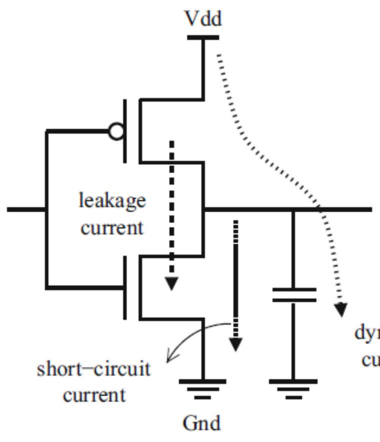


Figure 2: The dynamic, short circuit and leakage power components of transistor power consumption

where α is the activity factor ($0 \leq \alpha \leq 1$), i.e. a proportional constant indicating the percentage of the system that is active or switching, C is capacitance, V is the supply voltage, and f is the clock frequency. The activity factor is used to model the average switching activity in the circuit, which depends on the pattern of task which is executed on the CPU.

Since the dynamic power component dominates the total CPU power consumption, in our project, we simplify the model of CPU total power consumption as:

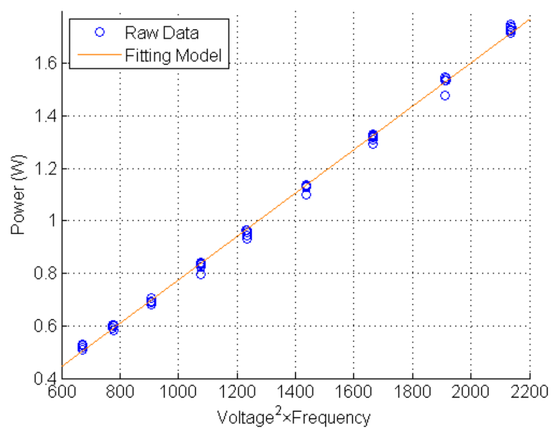
$$P_{total} = \alpha C V^2 f + \beta \tag{3}$$

The task of CPU power estimation comes down to obtaining good estimates for parameter αC as a whole and parameter β. We execute matrix multiplication program with several different input data size on the Odroid XU+E platform and collect 54 sets of data for big CPU and 48 sets of data for little CPU, including CPU power, supply voltage and CPU frequency. Here, we consider the product of CPU frequency and the square of supply voltage as the only predictor and consider CPU power consumption as response. Then, we use 10-fold cross-validation to train these data and then obtain a linear regression model, which would find estimates of the parameters so that the sum of the squared errors is minimized for big CPU and little CPU respectively. Figure 3 shows the big CPU power model and the small CPU power model we build for matrix multiplication task. The resulting power models for the big CPU core and the little CPU core are shown respectively in equations 4, 5.

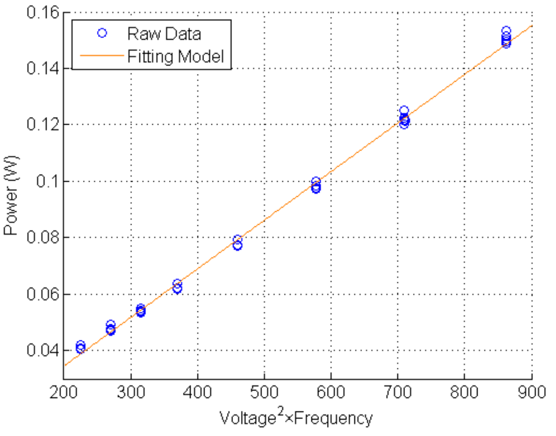
$$Power = 8.27 \times 10^{-4} V^2 f - 5.29 \times 10^{-2} \tag{4}$$

$$Power = 1.722 \times 10^{-4} V^2 f + 6.534 \times 10^{-5} \tag{5}$$

In addition, a certain CPU frequency have a demand of a certain amount supply voltage. In order to make our proposed system work, we also explore the correlation between frequency and supply voltage for big CPU and little CPU respectively. Figure 4a shows that there is a good linear relationship between supply voltage and frequency for big CPU. Figure 4b shows that, for little CPU, supply voltage

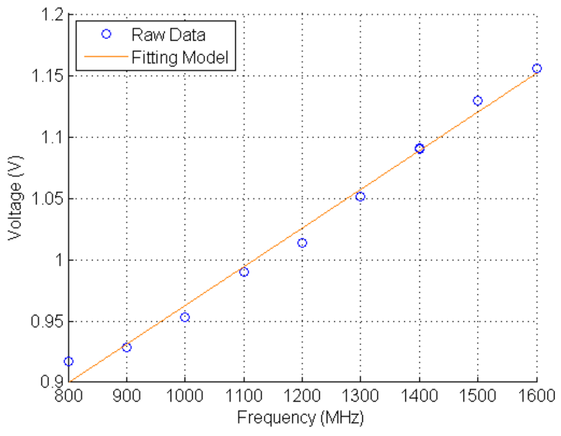


(a) Big CPU

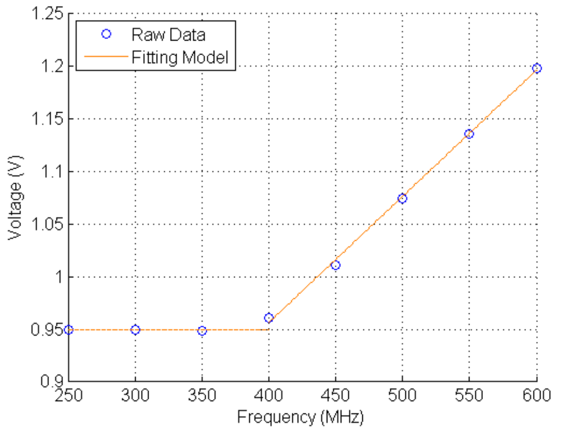


(b) Little CPU

Figure 3: CPU power models (matrix multiplication)



(a) Big CPU



(b) Little CPU

Figure 4: Correlation between frequency and voltage

is a constant value when frequency is lower than 400 MHz and then supply voltage linearly increase with the increased frequency.

The relation between voltage and frequency can be expressed in the following equations.

$$V = 0.95, 250MHz \leq f \leq 350MHz \tag{6}$$

$$V = 1.2 \times 10^{-3} f + 0.4765, 400MHz \leq f \leq 600MHz \tag{7}$$

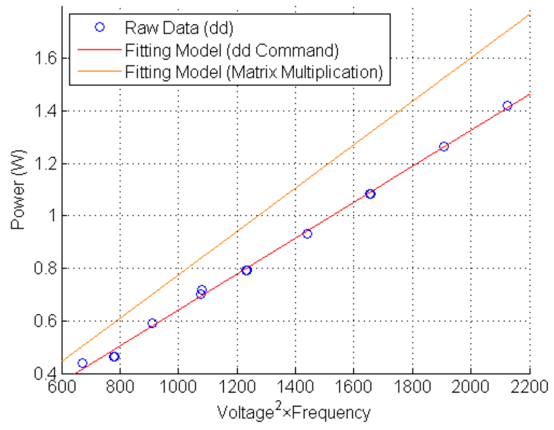
$$V = 3.152 \times 10^{-4} f + 0.6473, 800MHz \leq f \leq 1.6GHz \tag{8}$$

Furthermore, in order to observe whether the CPU power model is independent of the type of task which is executed on the CPU, we use the same method described above to build the big CPU power model and the little CPU model for dd command which is used to convert and copy a file in Linux operating system. Figure 5 shows the power model we build for dd command and also the comparison between the power models for two kinds of tasks, i.e. dd command and matrix multiplication. From Figure 5, we found that the slope of two straight line is different. This is because different kinds of tasks have different switching activity in the circuit level hence the corresponding activity factor α is different. In conclusion, the type of task can influence the parameters in the CPU model. This motivates us to develop an adaptive CPU power model for the Odroid-XU+E platform.

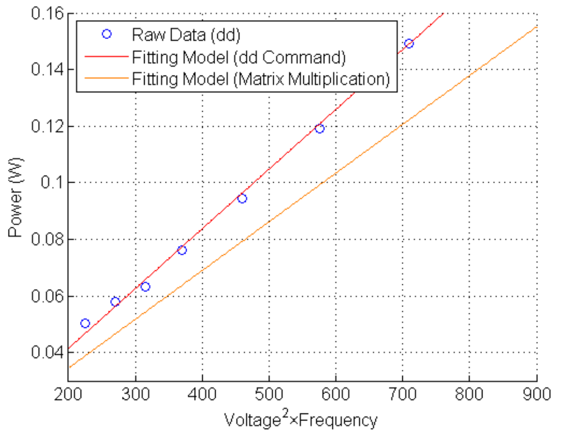
3.2 Adaptive Power Model

Power models are used in many different applications. However, power values calculated from power models are mere estimates. As shown in the comparison between the power consumption of a matrix multiplier and that of dd command, the activity factor α varies depending on different types of tasks. However, it is hard to predict α. From our experience building a power model, α multiplied with C can be obtained as the slope of the regression line of the power model. However, training separate models for different tasks to find α is not very feasible for most cases because it requires huge data sets beforehand.

We propose an adaptive power model to overcome such difficulty. The adaptive power model we have designed is inspired by the general feedback control system. Figure 6 is the overview of the adaptive power model design. The adaptive power model first imports the original power model we have built in subsection 3.1. From a user space CPUfreq



(a) Big CPU



(b) Little CPU

Figure 5: CPU Power models (dd command)

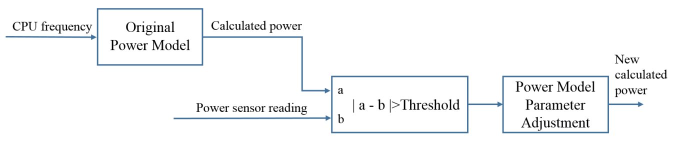


Figure 6: Adaptive Power Model

interface, "scaling_cur_freq", the current frequency is obtained. Then, voltage is computed from the current frequency. These values along with predefined parameters, α C and β, are used to calculate the power consumption.

Then, the adaptive power model generation algorithm collects the power sensor data for a very short time period. The measured power is then compared with the calculated power from the original model by taking the absolute value of the difference between these power values. The absolute value is then compared with a threshold. The threshold value depends on the CPU core. For big CPU cores, we have a predefined threshold taken from the largest residual error obtained when we trained the big CPU core power model. The threshold for little CPU cores are taken from the largest error obtained when we trained the little CPU core power model. If the difference between the measure power and calculated power is greater than the threshold, the parameters αC are re-calculated with equation 9.

$$\alpha C = \frac{P_{measured} - \beta}{V^2 f} \tag{9}$$

The new αC value is used to adjust the power model and the algorithm calculates the power with adjusted model. The new calculated power can be compared with the sensor reading for further parameter adjustment.

To demonstrate the effectiveness of our adaptive power model, we have designed an energy analysis program that estimates the power consumption from the CPU frequency and predicts execution time based on the load and the CPU frequency. Then, the energy is computed from the resultant values. The program is implemented with our power model, execution time model, and adaptive power model as shown in Figure 7. The execution time model is explained in the following subsection. The program's adaptive power model algorithm adjusts all parameters instantly as a specific task starts running.

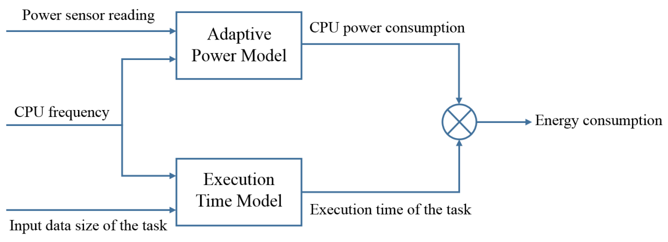


Figure 7: Energy consumption prediction

By integrating our adaptive power model algorithm, many applications that need power estimation via power models can have only "one" model that adapts to different types of tasks.

3.3 Execution Time Model

In our project, we define the execution time of a program as the time that elapses from when the first CPU starts executing on the problem to when the last CPU completes execution. Actually, it is quite impossible to exactly predict the execution time of an arbitrary program on a specific computational resource. To avoid such complexity, we only chose programs that will have predictable tasks. In our project, we built two sets of execution time models for a matrix multiplication program and Linux dd command. To demonstrate our approach in this section, we focus on a specific task, i.e. matrix multiplication. We execute the matrix multiplication program with different input matrices dimensions under each frequency and collect 272 sets of data. Figure 8 is a three dimensional visualization plot of our collected raw data. From Figure 8, we have confirmation that there is indeed a relationship between execution time and CPU frequency as well as dimensions of two input matrices. In order to obtain the quantized models, in the following subsection, we will explore two kind of correlations respectively: the correlation between execution time and CPU frequency as well as the correlation between execution time and input data size.

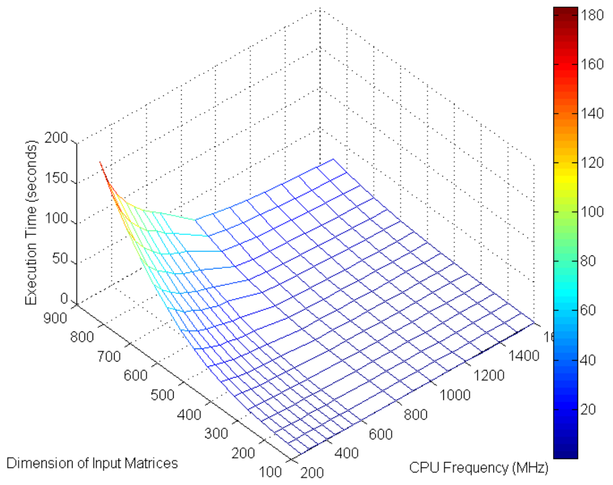


Figure 8: Execution Time Raw Data Visualization

3.3.1 Correlation between Execution Time and CPU Frequency

While the CPU core is a major contributor to a system's power consumption, other subsystems, such as memory and I/O, are also significant and can in some cases even dominate the CPU. Moreover, such contributions generally are independent of the core frequency. In this perspective, Snowdon et al.[6] modeled the overall execution time T as a function of the various clock rates f_x used in the system:

$$T = \frac{C_{cpu}}{f_{cpu}} + \frac{C_{bus}}{f_{bus}} + \frac{C_{mem}}{f_{mem}} + \frac{C_{io}}{f_{io}} + ... \tag{10}$$

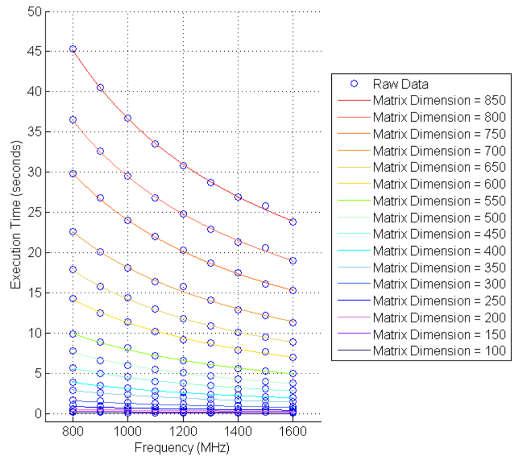
The coefficients C_{cpu}, C_{bus}, C_{mem}, C_{io}... depend on the instruction stream of the actual workload.

In our project, since memory and I/O subsystem will not dominate the CPU, we simplify the correlation between execution time and CPU frequency as:

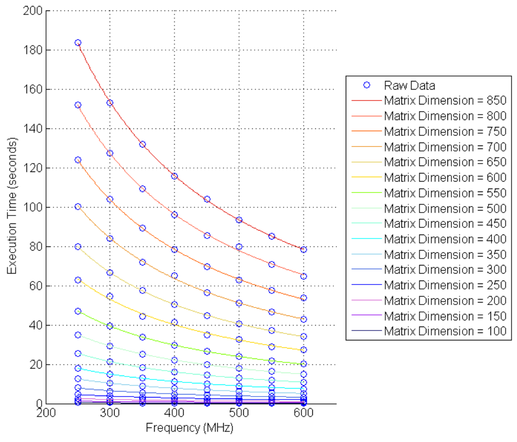
$$T = \frac{\alpha_i}{f} + \beta_i \tag{11}$$

where f is CPU frequency, α and β is the coffecients which are influenced by input matrices dimensions. In order to estimate parameters α and β for different workload, we run several simulations on the Odroid XU+E platform and collect 144 sets of data for Big CPU and 128 set of data for Small CPU, including execution time, CPU frequency and dimension of input matrices. Here, we consider the reciprocal of CPU frequency as predictors and consider execution time as response. Then, we use 10-fold cross-validation to train the data and then, obtain a linear regression model which would find estimates of the parameters so that the sum of the squared errors is minimized for each different workload. Figure 9 shows the correlation between execution time and the CPU frequency.

However, the above model is not workable on Odroid XU+E platform in practice, since the coefficient α and β in the model is depend on instruction stream of actual workload but Odroid XU+E platform doesn't have performance monitoring counters which is used for workload characterisation. Actually, we also try to build a regression model to predict the instruction stream of actual workload by using input matrices dimension as predictor, but the regression model we build has a low prediction accuracy. We leave the study of applying model that characterizes the relationship between execution time and CPU frequency to Odroid XU+E platform in practice to future work.



(a) Big CPU



(b) Little CPU

Figure 9: Correlation between execution time and frequency

3.3.2 Correlation between Execution Time and Input Data Size

In this subsection, we change to another perspective to explore execution time model. By manually analyzing source code of matrix multiplication program, we know the time complexity of each part in the program, which provides us a bound function depending on the input data size. The time complexity of basic matrix multiplication part is $O(n^3)$, and the time complexity of input matrix initialization part and multiplication result output part is $O(n^2)$, where n is the dimension of two input matrices. As a consequence, the estimated execution time of matrix multiplication program can be modeled as a function of the dimension of two input matrices:

$$T = \alpha_{f_i} n^3 + \beta_{f_i} n^2 + \gamma_{f_i} \tag{12}$$

where α and β represent estimated coefficients for predictors, namely n^3 and n^2 respectively, γ represents the estimated intercept, and n is the dimension of two input matrices.

In order to estimate parameters α, β and γ for each frequency, we run several simulations on the Odroid XU+E platform and collect 272 sets of data (16 sets of data, including execution time and input matrices dimension, for 17 different frequency). Here, we consider the cube of input matrices dimension and the square of input matrices dimension as predictors and consider execution time as response. Then we use 10-fold cross-validation to training the data and then obtain a linear regression model which would find estimates of the parameters so that the sum of the squared errors is minimized for each frequency. Figure 10 shows the correlation between execution time and the dimension of input matrices. In our project, we implement this execution time model which characterize correlation between execution time and input matrices dimension for each available frequency of Samsung Exynos 5 Octa 5410 on the Odroid XU+E platform.

4. IMPLEMENTATION

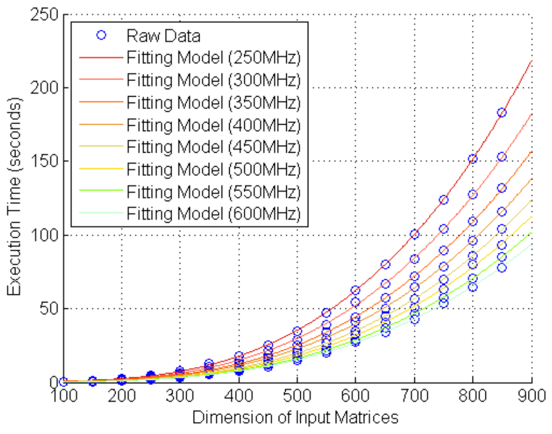
We implemented tools for data collection and model training. From the resultant power model and data collecting method, we were able to implement an adaptive power model, which automatically adjusts the model by correcting the activity factor based on real power sensor data. The data collection tool and the adaptive power model are implemented in C and C++ on the ODROID-XU+E running Linux kernel 3.4.75. On the other hand, the model training is implemented with R script.

4.1 Data collection

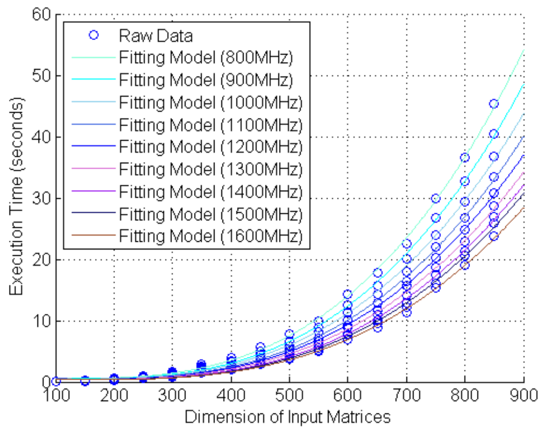
Data collection tools utilize the power/voltage sensor interfaces, CPUfreq subsystem, and some Linux functions. The Power and voltage data were collected with different CPU frequencies and different workload and were used to train a power model. The execution time data are used for the execution time model, which helps analyze energy consumption on the platform.

4.1.1 Power and voltage data

The biggest problem with the power/voltage sensors integrated onto the ODROID-XU+E is that only one sensor is connected to all A7 cores and another sensor is connected to all A15 cores. Thus, only sum of all A7 cores' power and sum of all A15 cores' power can be collected. Since the load on each core will not be uniformly distributed, it is hard to know the power dissipation on each core. In order to overcome such issue, we deactivated all other CPU core pairs except for one pair (cpu0). Deactivation of CPU core pairs can be easily done in the user space as shown in Figure 11. With only one cpu core active, the power and voltage data apply to just one core. Throughout the implementation of all tools and systems, we had kept only one CPU core pair active. We implemented the power/voltage data collection tool in C. It manually selects from the lowest available frequency, 200MHz, to the highest available frequency, 1.6GHz.

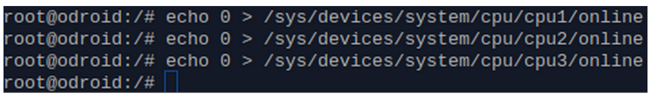


(a) Litte CPU



(b) Big CPU

Figure 10: Correlation between execution time and the dimension of input matrices



```
root@odroid:/# echo 0 > /sys/devices/system/cpu/cpu1/online
root@odroid:/# echo 0 > /sys/devices/system/cpu/cpu2/online
root@odroid:/# echo 0 > /sys/devices/system/cpu/cpu3/online
root@odroid:/# 
```

Figure 11: Deactivation of CPU cores in Linux

For each frequency, we used fork() function to create a child process that runs a measurement algorithm and the parent process makes a system call via system() to run a one-task-specific program. When system() returns, the program is done running and the parent process sends an interrupt signal to the child process to stop the measurement. The measurement algorithm is implemented with a while loop. We used userspace power/voltage sensor data interfaces. In every loop, the sensor reading data are read from the interfaces and timestamped. These data are appended to an output text file for each CPU frequency. We are aware of the overhead of this power/voltage measurement tool. To reduce the overhead, the small sampling rate is chosen. We made data collection sleep for 0.1 second for every loop. Therefore, sampling rate is only 10Hz. Since the power/voltage do not change significantly over time, very low sampling rate is still suitable for data collection. Then, the collected data are compared with the voltage and power displayed directly from the sensor interfaces in real-time while the specified program is running in the background. The difference is negligible, so the data collection tool is accurate enough for our purpose.

4.1.2 *Execution time data*

The execution time data collection tool is simply a modification of the power/voltage data collection tool. For each selected CPU frequency, it makes a system call via system() to run a specified program. Just before the system call, it gets a timestamp using gettimeofday() function. Then, it gets another timestamp after system() returns. The difference between these timestamps is the approximate execution time. To verify this method, we ran ”dd if=/dev/zero of=/dev/null count=10000000”. At the end of its execution, it prints out the elapsed time to the stdout. We compared the measured execution time and the printed time by the program itself. The difference is negligible, so the execution time data collection tool is accurate enough for our purpose.

4.2 Model training

The raw training data are collected by using the methods mentioned in the previous subsections and then, are stored in several csv format files. The power model training and the execution time model training are all implemented in R scripts. In R scripts, we used read.csv() function to read raw data from the csv format files and then, generated data, frame which is used for data pre-processing. In the data pre-processing part, we subset the generated data frame to obtain predictor and prediction target that we want to use for model training. We used the train() function in R package caret to train robust linear regression model. The train() function generates a resampling estimate of performance. Because the training set size is not small, 10-fold cross-validation should produce reasonable estimates of model performance. The trainControl() function is used to specify the type of resampling. Finally, the training results, i.e. the model parameters, were visualized by using Matlab.

4.3 Adaptive power model

As mentioned in subsection 3.2, we implemented an energy consumption predictor program to demonstrate adaptive power model. This program accepts two input arguments, which are dd count size and the required minimum execution time, i.e. QoS(Quality of Service). The program is written in C.

First, it calculates execution time with the execution time model and the dd count size given as an input. The resultant execution times for each available are stored in an array. The elements of the array are then compared with the QoS specified by the user. From this, we can find the minimum CPU frequency that satisfies the QoS. With a for loop, it iterates from the minimum required frequency to the maximum frequency and calculates power consumption for each frequency with the default power model using $\alpha = 8.27 * 10^{-4}$ for the big CPU core and $\alpha = 1.722 * 10^{-4}$ for the little CPU core. In each iteration, the power is multiplied with the execution time to calculte energy consumption. The results are stored in an array. The assumption on the energy calculation is that the power dissipation is constant for the same workload and the CPU frequency. Therefore, the integral of the power over execution time is just the product of these two. Then, it finds the minimum value of the energy consumption array and print a message suggesting which frequency would have most energy saving.

Then, we run the same simulation with the adaptive power model. fork() is used to create a child process that runs ”dd if=/dev/zero of=/dev/null count=10000000”. A large count number is chosen on purpose to ensure that it is running for sufficient time. The parent process sets the CPUfreq governor to Userspace governor and select a frequency that will choose the little CPU core. Then, it sleeps for 2 seconds to avoid power calculation before the child process starts running the dd command. The adaptive power model algorithm instantly calculates the new αC value for the little CPU core. Then, a frequency that will choose the big CPU core. Then, the algorithm calculates the new αC value for the big CPU. With these new values, it adjusts the power model and repeats the energy consumption calculation algorithm described above.

5. RESULTS AND EVALUATION

There are several approaches we have tried to evaluate our work. These evaluation approaches are discussed in the subsections.

5.1 Power model

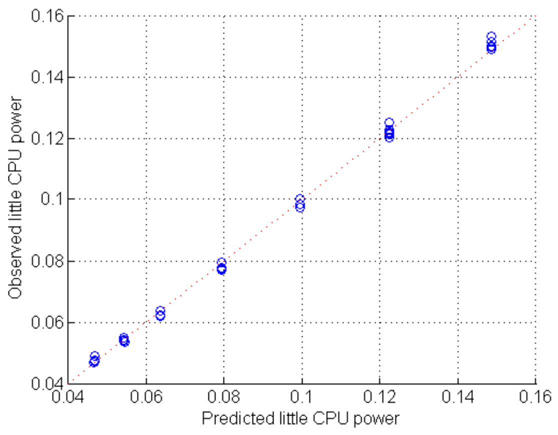
To evaluate the prediction accuracy of our CPU power models, we compared our predicted CPU power data with the observed power data obtained from power sensor on the Odroid-XU+E platform. Figure 12 shows observed CPU power values versus predicted CPU power values for the big CPU power test set and the little CPU power test set respectively. From Figure 12, we can find that all the points are around the straight line y=x, which indicates that the predicted CPU power values are all very closed to their corresponding observed CPU power values. As a consequence, our big CPU power model and small CPU power model are both accurate enough.

5.2 Execution time model

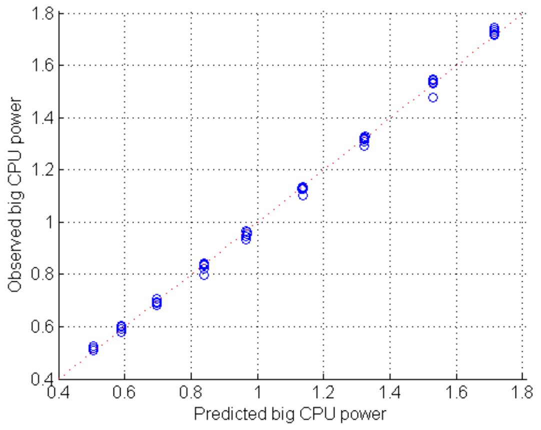
We can directly evaluate the execution time model though Figure 9 which characterizes the correlation between execution time and CPU frequency as well as Figure 10 which characterizes the correlation between execution time and input data size. From Figure 9 and Figure 10, we can see that all the raw data points are closed to their corresponding fitting model lines, which is enough to demonstrate that our execution time models have a good performance.

5.3 Adaptive power model

As described in subsection 3.2, we designed a simple program that displays power, execution time, and energy computed with our models. The purpose of this program is to suggest a frequency that would minimize the energy consumption. However, we use it to evaluate and demonstrate the effectiveness of the adaptive power model algorithm. The program first displays the execution time, the power dissipation, and the energy consumption computed using our power model and execution time model. Then, it displays another set of these items but using our adaptive model. Figure 13 is an output when the processor runs ”dd



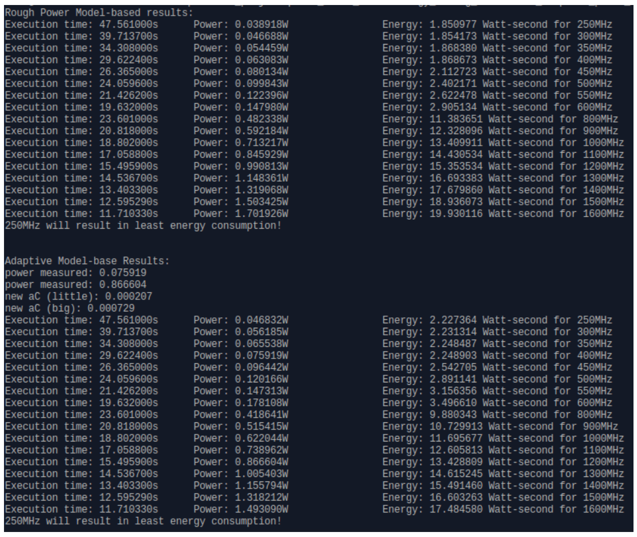
(a) Little CPU



(b) Big CPU

Figure 12: CPU power model evaluation

if=/dev/zero of=/dev/null count=10000000". The program



```
Rough Power Model-based results:
Execution time: 47.561000s      Power: 0.038918W                Energy: 1.850977 Watt-second for 250MHz
Execution time: 39.713700s      Power: 0.046688W                Energy: 1.854173 Watt-second for 300MHz
Execution time: 34.308000s      Power: 0.054459W                Energy: 1.868380 Watt-second for 350MHz
Execution time: 29.622400s      Power: 0.063083W                Energy: 1.868673 Watt-second for 400MHz
Execution time: 26.365000s      Power: 0.080134W                Energy: 2.112723 Watt-second for 450MHz
Execution time: 24.059600s      Power: 0.099843W                Energy: 2.402171 Watt-second for 500MHz
Execution time: 21.426200s      Power: 0.122396W                Energy: 2.622478 Watt-second for 550MHz
Execution time: 19.632000s      Power: 0.147980W                Energy: 2.905134 Watt-second for 600MHz
Execution time: 23.601000s      Power: 0.482338W                Energy: 11.383651 Watt-second for 800MHz
Execution time: 20.818000s      Power: 0.592184W                Energy: 12.328096 Watt-second for 900MHz
Execution time: 18.802000s      Power: 0.713217W                Energy: 13.409911 Watt-second for 1000MHz
Execution time: 17.058800s      Power: 0.845929W                Energy: 14.430534 Watt-second for 1100MHz
Execution time: 15.495900s      Power: 0.990813W                Energy: 15.353534 Watt-second for 1200MHz
Execution time: 14.536700s      Power: 1.148361W                Energy: 16.693383 Watt-second for 1300MHz
Execution time: 13.403300s      Power: 1.319068W                Energy: 17.679860 Watt-second for 1400MHz
Execution time: 12.595290s      Power: 1.503425W                Energy: 18.936073 Watt-second for 1500MHz
Execution time: 11.710330s      Power: 1.701926W                Energy: 19.930116 Watt-second for 1600MHz
250MHz will result in least energy consumption!


Adaptive Model-base Results:
power measured: 0.075919
power measured: 0.866604
new aC (little): 0.000207
new aC (big): 0.000729
Execution time: 47.561000s      Power: 0.046832W                Energy: 2.227364 Watt-second for 250MHz
Execution time: 39.713700s      Power: 0.056185W                Energy: 2.231314 Watt-second for 300MHz
Execution time: 34.308000s      Power: 0.065538W                Energy: 2.248487 Watt-second for 350MHz
Execution time: 29.622400s      Power: 0.075919W                Energy: 2.248903 Watt-second for 400MHz
Execution time: 26.365000s      Power: 0.096442W                Energy: 2.542705 Watt-second for 450MHz
Execution time: 24.059600s      Power: 0.120166W                Energy: 2.891141 Watt-second for 500MHz
Execution time: 21.426200s      Power: 0.147313W                Energy: 3.156356 Watt-second for 550MHz
Execution time: 19.632000s      Power: 0.178108W                Energy: 3.496610 Watt-second for 600MHz
Execution time: 23.601000s      Power: 0.418641W                Energy: 9.880343 Watt-second for 800MHz
Execution time: 20.818000s      Power: 0.515415W                Energy: 10.729913 Watt-second for 900MHz
Execution time: 18.802000s      Power: 0.622044W                Energy: 11.695677 Watt-second for 1000MHz
Execution time: 17.058800s      Power: 0.738962W                Energy: 12.605813 Watt-second for 1100MHz
Execution time: 15.495900s      Power: 0.866604W                Energy: 13.428809 Watt-second for 1200MHz
Execution time: 14.536700s      Power: 1.005403W                Energy: 14.615245 Watt-second for 1300MHz
Execution time: 13.403300s      Power: 1.155794W                Energy: 15.491460 Watt-second for 1400MHz
Execution time: 12.595290s      Power: 1.318212W                Energy: 16.603263 Watt-second for 1500MHz
Execution time: 11.710330s      Power: 1.493090W                Energy: 17.484580 Watt-second for 1600MHz
250MHz will result in least energy consumption!
```

Figure 13: Energy consumption prediction output

runs only one iteration for the parameter adjustment. In

Calculated Power		
Trained Model	Adaptive Model	Percent Error
0.049211	0.046832	-4.83%
0.058114	0.056185	-3.32%
0.063152	0.065538	3.78%
0.075814	0.075919	0.14%
0.094412	0.096442	2.15%
0.119130	0.120166	0.87%
0.149021	0.147313	-1.15%
0.182674	0.178108	-2.50%
0.440806	0.418641	-5.03%
0.491711	0.515415	4.82%
0.590648	0.622044	5.32%
0.702031	0.738962	5.26%
0.822303	0.866604	5.39%
0.962241	1.005403	4.49%
1.112906	1.155794	3.85%
1.265207	1.318212	4.19%
1.441543	1.493090	3.58%

Table 2: Comparison of power calculated with a separately trained model and that with our adaptive model

other word, only one power measurement for each core is used for parameter adjustment. Table 2 shows the comparison between the power calculated with a separately offline trained model and the power calculated with our adaptive model. The lowest percent error is as low as 0.14% while the largest percent error is 5.39%. This is a significant accuracy even with only one parameter adjustment iteration. Multiple iterations can improve the accuracy even further.

Another interesting observation from the output of this program is that the energy consumption is the least with low frequency. In some cases, the suggested frequency is 250MHZ and in another cases, it is 300MHz. The theory on power and energy consumption claims that the energy consumption is independent of the CPU frequency. However, the output results show that energy consumption significantly changes. Our assumption is that the reason is the non-linearity of the execution time over frequency. The execution time also depends on the memory access speed, the bus speed, etc. Thus, increasing only CPU frequency can make other components as the bottleneck of the performance.

6. LIMITATIONS AND DISCUSSION

Our work focuses on a new approach to build a power model specifically for each type of task. The design and implementation have room for further improvements. In this section, we discuss about the limitations of our works and propose future work according to these limitations.

First, we are forced to use only one CPU core pair. As mentioned earlier, this limitation is largely due to the limited number of power monitor sensors. There are only one sensor for all A7 cores and one sensor for all A15 cores. If there are eight sensors for each core, we may allow all CPU core pairs to operate and collect data for each core. In the scope of our work, using only one CPU core pair is, in fact, not a problem since all CPU core pairs are identical.

However, Capability of power/voltage measurement on each core would allow to investigate real-case power consumption of multi-core processors.

Second, the accuracy of the data collection tools can be improved. Based on our observation on the collected data, there are peaks on both power data and voltage data. The potential causes for such peaks are not investigated. For analysis, we chose the mean values of both power data and voltage data. However, modes are more suitable in this case because a mode is less sensitive to peak data.

Third, the adaptive power model can adjust the parameters only when the specific task is currently running. The algorithm can be modified to accepted past power sensor readings from log files. although this modification can allow the adaptive power model to work for past and present tasks, but it cannot predict power consumption of a task that has never run on the platform.

Forth, our adaptive power model may not be suitable for the tasks that runs for very small time period. The power sensors on ODROID-XU+E delivers the data to the processor via I2C bus. There must be a delay associated with the bus.

Fifth, the overhead of the adaptive power model is not evaluated. The overhead largely depends on how often the algorithm adjusts the parameters. In the energy consumption predictor program demo, only one specific task runs so that it was not necessary for the adaptive model algorithm to adjust often. However, if we have a program that consists of a large number of little tasks, the adaptive model must be capable of fast adjustment. Fast adjustment will have a significant overhead. The trade-off of fast adjustment and low overhead should be studied.

7. CONCLUSIONS

This paper proposed power models for Samsung Exynos 5 Octa 5410 at the CPU level. The behavior of the CPU migration on this big.LITTLE architecture is investigated and separate models for the big core and the little core are generated. The adaptive power model is proposed to improve the accuracy, which suffers from the activity factor variation for different tasks. With only one parameter adjustment iteration, the accuracy of the model with respected to offline trained model is approximately 0.14% -5.39% . With multiple parameter adjustment iterations, the accuracy will be improved. The scope of our work is not generating state-of-art power models but providing a new approach for accurate power model generation. There is still a big room for improvement. We expect that system development that requires accurate CPU-level power model would benefit from our approach with future improvement.

8. ACKNOWLEDGMENTS

The authors would like to thank Professor Mani Srivastava.

9. REFERENCES

[1] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Robert P. Dick, Z. Morley Mao, Zhaoguang Wang, and Lei 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.*

[2] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. Heterogeneous multi-processing solution of exynos 5 octa with arm big.little technology.

[3] Madhu Palmur, Zhichao Li, and Erez Zadokg. Cpuidle from user space.

[4] P.R Panda, B.V.N Silpa, A. Shirivastava, and K. Gummidipudi. *Power-efficient System Design.* Springer, 2010.

[5] Vishwani D.Agrawal and Srivaths Ravi. Dynamic and static power in cmos. 2007.

[6] David C. Snowdon, Godfrey Van Der Linden, Stefan M. Petters, and Gernot Heiser. Accurate run-time prediction of performance degradation under frequency scaling. *Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2007.