Profiling and Understanding CPU Power Management in Linux

Ti Zhou*, Haoyu Wang†, Xinyi Li‡ and Man Lin§
Department of Computer Science, St. Francis Xavier University, Canada
Email: *x2019cwm@stfx.ca, †x2020fcw@stfx.ca, ‡x2021gim@stfx.ca, §mlin@stfx.ca

***Abstract*—Dynamic Voltage and Frequency Scaling (DVFS) is a popular technique for power management. Understanding the DVFS principles and the current practice of existing DVFS governors embedded in operating systems (OS) is essential. This work aims to provide a deep understanding of DVFS power management through real-time profiling on various Linux platforms, including an Intel-based laptop, an ARM-based Jetson Nano Board, and a Raspberry Pi platform. We first present the theoretical model for dynamic and static power consumption and describe experiments on three platforms to show how frequency affects their power consumption. We then visualize the real-time behaviour of the existing DVFS governors: the *Ondemand* and *Conservative* governors in OS kernels on multiple platforms under different parameter settings. Furthermore, we identify issues that may be encountered in actual hardware experiments but are often overlooked by researchers. Examples are coarse-grained frequency levels and the OS interface not reflecting the actual frequency. Finally, we design experiments to explore the relationship between utilization and power. This work can help DVFS algorithm designers to consider the practical aspect of integrating DVFS algorithms into actual systems.**

***Index Terms*—CPU power management, Dynamic Frequency Scaling, Real-time Profiling, OS governors**

I. Introduction

Power management is a crucial research topic in real-time systems. Though there is some research exploring power management on real devices [1] [2] [3] [4] [5] [6] [7] [8], a vast majority of the current research [9] [10] [11] [12] in power management for systems evaluate their power management algorithms based on empirical formulae and/or simulation. The interaction between kernel modules, the architecture of the different hardware platforms, and hardware compatibility together affect the power controlled by a software power management controller. The power on the actual devices is thus likely different from that of the simulated environment. In this work, we illustrate power management on actual devices (three different platforms) by profiling the Linux kernel to provide a deep understanding of existing governors and reveal some of the on-device power management engineering issues.

We start by showing how frequency affects real devices' static and dynamic power and performance. We first design experiments to show the power consumption of three platforms running with different frequencies with idle workload. We then illustrate the relationship between performance, power and frequency on the three platforms for a CPU-Intensive workload. We discuss the correlation between the commonly used empirical formulae used in the literature and the measured power of the actual devices through visualization.

We then visualize how the two commonly used dynamic power management governors embedded in Linux, (*Ondemand* and *Conservative*), behave on real devices. We do this by showing the real-time profile of the CPU load and the governor's selected CPU frequency. We then explain how the governor works based on the algorithms summarized from Linux governor manuals and justify the behaviour shown in the profiled charts. In the *Ondemand* governor, there is a parameter called *powersave_bias* which can be set by users. We showed how this parameter affects the performance of the *Ondemand* governor on real testing platforms.

Then, we design experiments to identify the frequency setting discrepancy (i.e. inconsistencies between system settings and real results) on different platforms.

Finally, we design experiments to explore how utilization affect power consumption.

Our previous research designed power management algorithms and evaluated them on real devices [13] [14] [15]. This work aims to provide researchers with experimental designs to show how DVFS and commonly used Linux power management governors work. Experiments were designed to show factors that might affect the DVFS performance on real devices. We hope the practical engineering aspects discussed in this paper and how we visualize the real-time behaviour of DVFS governors will help researchers deploy and evaluate their DVFS algorithms on real devices.

II. Power Consumption Model

Power consumption model has been explored by different researchers. Some work built a power model [16] [17] for the entire system considering the power consumption of the CPU and other components. Several works [18] [19] [20] [21] have explored the relationship between the power model and some parameters in proposing their own energy-saving methods.

Before designing a policy to adjust voltage/frequency to save energy, a basic understanding of how the power consumption model works on the actual hardware enables the researchers to make a sound judgement on the reasonableness of the results. This section is intended to help understand the relationship between voltage/frequency and power consumption by designing experiments to show some of the relations through power measurement on three real platforms.

The CPU power consumption is modelled based on the CMOS circuits model, described in detail in [22]. The total power consumption P is made up of two types of power consumption: the dynamic and static power consumption, shown in Equation 1.

$$P = P_{dynamic} + P_{static} \tag{1}$$

A. Dynamic Power Consumption

Dynamic power consumption comes from capacitor charging and discharging (switching power) and from circuit short-circuiting caused by circuit flip-flopping (short-circuit power). Thus, the expression for dynamic power consumption can be defined as Equation 2 [22].

$$P_{dynamic} = P_{switch} + P_{short} \tag{2}$$

As the short-circuit power consumption accounts for only a small fraction of the dynamic power consumption [23], it is assumed that the dynamic power consumption is approximately equal to the switching power consumption, defined in Equation 3, where α is called the activity factor, C is the load capacitance, V_{dd} is the supply voltage and f is the clock frequency [22].

$$P_{dynamic} \approx P_{switch} = \alpha C V_{dd}^2 f \tag{3}$$

In addition, V_{dd} and f have the following relationship [22].

$$f \propto \beta(V_{dd} - V_{th})^2 / C V_{dd} \tag{4}$$

The V_{th} refers to the threshold voltage, the voltage in the input voltage that causes the input current to change sharply, which is much less than the supply voltage. β is a constant which is approximately equal to 1. Thus, we can assume that f and V_{dd} have the following approximate proportional relationship.

$$f \propto V_{dd} \tag{5}$$

Therefore, based on Equation 3 and Equation 5, we can obtain the following relationship between dynamic power consumption and voltage/frequency.

$$P_{dynamic} \propto V_{dd}^3 \propto f^3 \tag{6}$$

As a result, the dynamic power consumption varies in a cubic relationship with the voltage/frequency.

TABLE I
THE PARAMETERS OF THE THREE PLATFORMS

	OS	CPU	Measurement Tools
Nvidia Jetson Nano Board 2GB	Linux 4.9	ARM Cortex-A75	Power Meter
Raspberry Pi 4B	Linux 5.15	ARM Cortex-A72	Power Meter
Laptop (Thinkpad T470)	Linux 5.4	Intel I5-7200U	Intel RAPL

B. Static Power Consumption

Static power consumption is much simpler than dynamic power consumption. It can be defined as Equation 7, where V_{dd} refers to the supply voltage and I_{static} refers to the static current [22].

$$P_{static} = V_{dd} \times I_{static} \tag{7}$$

The static current is the current consumed by the device itself when there is no workload, which means the device is not operating (CPU in idle state) but has the supply voltage. There are many sources of static currents, such as the sub-threshold leakage current. Thus, when reducing the supply voltage and static current, static power consumption is also reduced [22].

C. Observations of Power Consumption on Real Platforms

In this subsection, we show our experiments designed to observe how the dynamic and static CPU power consumption models reflect in real platforms. Experiments were conducted on the following three platforms: Jetson Nano Board 2GB (JTN), Raspberry Pi 4B (RBP), and a laptop (ThinkPad T470). We chose JTN and RBP because these two boards are popular on embedded platforms. The Thinkpad choice is used as a laptop comparison. The detailed information and configuration are shown in Table I. On the laptop, the energy consumption was measured by the RAPL interface [24]. While on the two embedded devices, the energy consumption was measured by a power meter, as these two embedded devices do not have an API similar to RAPL interface to read energy consumption directly.

The laptop supports a frequency range of 1.4 GHz to 2.5 GHz, which is verified in Section IV-B. The two boards support the frequency range from 0.102 GHz to 1.479 GHz and 0.6 GHz to 1.5 GHz, respectively.

1) Observations of Static Power Consumption: To observe static power, we let the device enter an idle state (CPU not performing any work) under each supported frequency. We then measure the energy consumption of the idle state during a given period.

For the laptop setting, $intel\_pstate$ is disabled and ACPI architecture is used. For the settings of $cpuidle$ subsystem, the $disable$ value of all idle states for each CPU core is set to 0, which allows the CPU enter those idle states. Similar experiments are done in the JTN and the RBP.

For the board devices (JTN and RBP), we conducted two sets of experiments for static power measurement. In the first set of experiments, the board is connected with only the power and network cables. In the second set of experiments, three peripherals are plugged into the board: a monitor, a keyboard and a mouse. The board's power consumption was also measured at each frequency. The results of the experiments are shown in Figure 1.

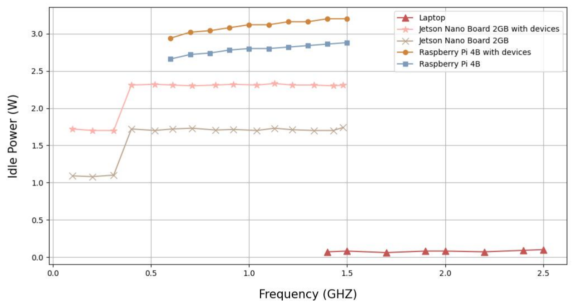

Fig. 1. Idle power in three platforms

We observed from Figure 1 that the static power consumption measured on the laptop is extremely low when it is in an idle state. In contrast, the other two devices have higher values of static power consumption with or without peripherals attached. Our explanation for this is that the CPU of the laptop supports more idle states than the other two devices, and when there is no workload, it can enter deeper idle states, which rarely consume energy. We conducted an experiment to verify our explanation. We set $disable$ value for deep $cpuidle$ states to 1 on the laptop so that its CPU cannot enter those deep idle states. In this case, the static power consumption measured became 2.1 W on average. The average static power measured before the deep idle states were enabled (shown in Figure 1) was only 0.1 W. This difference illustrates that the idle governor on the laptop is very powerful in contributing to energy saving. Thus, the readings from the RAPL interface reflect the energy saving effect both by the DVFS governor and the idle governor if $cpuidle$ states are not disabled.

Figure 1 shows that the JTN board's static power consumption has two distinct levels in both settings (with and without peripherals). The JTN has about the same static power consumption under the first three supported frequencies, while the rest supported frequencies share about the same static power consumption. The latter one is much higher than the former one. We inferred that the jump is due to voltage jump while the consistency is due to the same voltage.

By comparing the two cases of the JTN board (with and without peripherals) in Figure 1, the static power consumption in the case with peripherals is higher than that in the case without peripherals. The difference is due to the fact that accessing more peripherals will increase the static current, resulting in higher static power consumption. The experiment confirms the formula in Equation 7 that the static power consumption increases as the static current increases.

The static power consumption trend in RBP looks different from that of JTN board. RBP's static power consumption increases slowly with frequency. It does not have the same jump as the JTN because the gap between the two voltages on the Raspberry Pi is too small. RBP has two voltage levels. One is 0.835 V and another is 0.875 V. The voltage of 0.835 corresponds to the first frequency, while the frequency of 0.875 corresponds to all subsequent frequencies. The increase in power consumption from the first to the second voltage without the peripheral plugged in is 0.06 W. In comparison, the average increase in power consumption by changing the frequency while keeping the voltage constant is 0.02 W. The same trend can be found in the case of attached peripherals.

2) Observations of Dynamic Power Consumption: The following experiments were performed on three platforms to observe the dynamic energy consumption. The settings of $intel\_pstate$ and $cpuidle$ subsystem are the same as that in static power consumption experiments. A CPU-intensive workload (100% utilization for all cores) was run several times at each frequency supported by the platform, and the average running time and energy consumption were recorded. The experiment results are shown in Figure 2. These results are normalized with respect to the minimum value (the minimum value as 1).

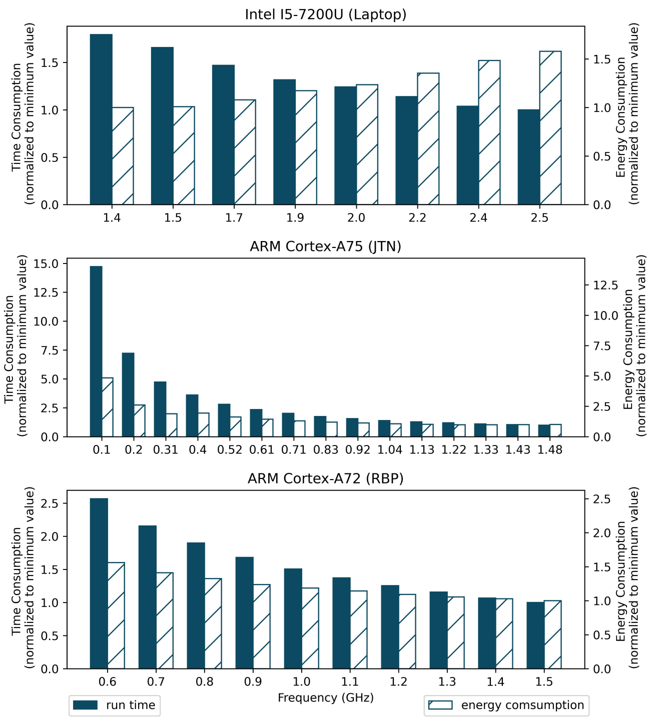

Fig. 2. Time consumption and energy consumption on three platforms

As can be seen from Figure 2, the running time decreases with frequency increases. The total energy consumption, however, shows different trends on each platform. On the laptop platform, energy consumption increases with frequency, while on the other two platforms, energy consumption decreases with frequency. This is because, on the laptop, there are many voltage/frequency levels. When frequency increases, the voltage also increases. This has led to an increase in both static and dynamic energy consumption.

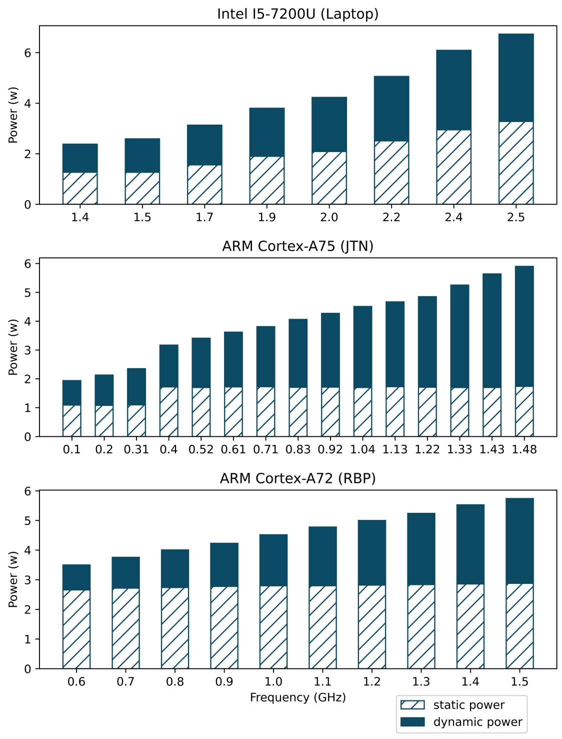

Fig. 3. Power consumption on three platforms

To better illustrate the dynamic and static power composition, we calculate the dynamic power consumption based on the static energy and the total energy. The results are shown in Figure 3, where the data for static power consumption is taken from Figure 1 except the laptop. For the laptop, instead of using the power shown in Figure 1 as the static power estimation, we use the power consumption measured with the *cpuidle* states disabled, as the laptop would not enter the idle state when the benchmark keeps running. While for the other two devices, we did not choose to use the data that *cpuidle* states disabled because there are only two *cpuidle* states on the board and they do not have much effect on energy. The difference between the data with *cpuidle* states enabled and disabled is only 0.06 W on average. Figure 3 confirms that dynamic energy consumption increases sub-squared with frequency. For both boards, the reason for the increase in dynamic power consumption but the decrease in total energy consumption as frequency increases is that, at low frequencies, there is more static power consumption than dynamic power consumption. The static power consumption remains the same, but static energy consumption decreases due to the decrease in time in completing the workload (note that the running time decreases with frequency increases). This decrease is much larger than the increase in dynamic energy consumption.

III. Analysis of *Ondemand* Governor and *Conservative* Governor

In this section, we describe the experiments that visualize the policies of the Linux built-in DVFS governors: *Ondemand*, and *Conservative* governors [25] with two benchmarks. One is *FaceAndAudioRecog* that we designed. This benchmark first runs a face recognition program which reads a photo and recognizes the location of each face in the image while recording audio simultaneously. After completing the face recognition program, the benchmark will sleep until 0.6 seconds. Then it starts to extract features from the recorded audio. The second benchmark is the *Basicmath* taken from Mibench [26]. It uses *SolveCubic()*, *usqrt()*, and *radtodeg()* functions to perform some basic math calculations. The benchmarks are run periodically, with 1.0 and 5.0 seconds as the task period for *FaceAndAudioRecog* and *Basicmath*, respectively.

Our profiler records the trace of the maximum load of all the cores, average load of all the cores and frequency of adjustment running these benchmarks using the governor. The recording of this information is done via file writes in the kernel. We conducted this set of experiments on the three platforms in Table I.

A. Analysis of Ondemand Governor

The *Ondemand* governor is an immediate response governor whose purpose is to translate CPU utilization into CPU frequency in the most immediate way. It performs a frequency adjustment after every period, determining the frequency of the next cycle based on the CPU load of the current period.

The profiling results for *Ondemand* governor are shown in Figure 4. In each subplot, the y-axis on the right side represents the load, and the y-axis on the left side represents each frequency level supported by the device. The vertical blue dashed line represents the task period. The green bar is the maximum CPU load in a time period, while the sloping bar is the average load of the four cores. The orange line represents the frequency chosen by the governor in a sampling point. It is worth mentioning that here the bar represents the maximum and average load between the two sampling points.

When running the *FaceAndAudioRecog* benchmark, the *ondmenad* governor shows the same trend on all three platforms. The highest frequency is selected when the maximum load on the CPU is high at the beginning of the run, and when the maximum load is reduced, the frequency selected is

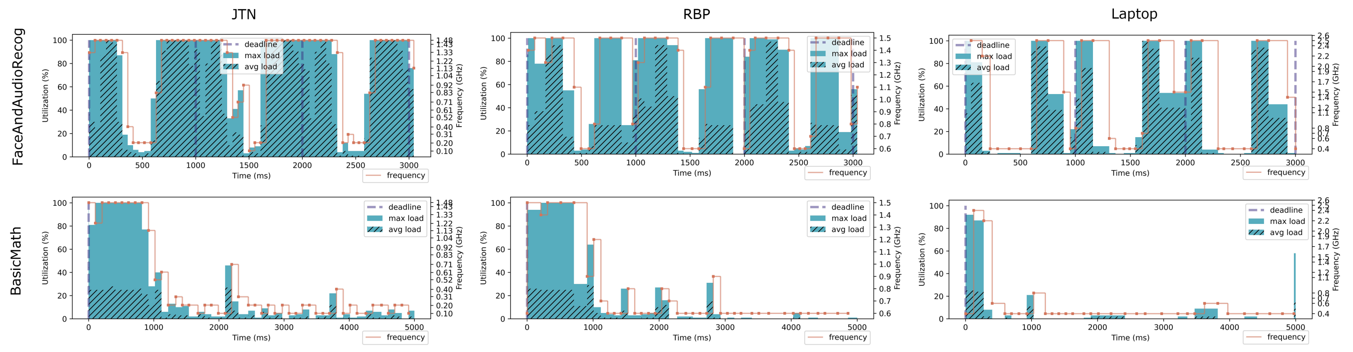

Fig. 4. The performance of *FaceAndAudioRecog* (the top row) and *Basicmath* (the bottom row) under *Ondemand* governor on three platforms.

also reduced. The maximum load in the previous time interval always determines the frequency in the next time interval. When the first half of the benchmark finishes running, and the CPU is in an idle state with little load, *Ondemand* will always select the lowest frequency. The second half starts with an increased frequency. Such a frequency selection is the same in every cycle. The workload behaviour of *Basicmath* is simpler, but it is still a good reflection of *Ondemand*'s policy.

The advantage of *Ondemand* governor is that the frequency can be adjusted upward in time to meet the performance requirements of the platform according to the utilization. For example, when the system runs the *FaceAndAudioRecog* benchmark, the *Ondemand* can quickly adjust to the maximum frequency in the second half when the benchmark wakes up from the sleep. At the same time, the frequency can be quickly reduced during the CPU idle time period to achieve energy savings. However, we can also observe that there is one unit time interval of delay in *Ondemand* in responding to the load change. As shown in Figure 4, there are many instances that the current load is extremely small, yet a high frequency is chosen because of the high load of the previous time period. This delay can lead to energy waste.

B. *Tune the powersave_bias of Ondemand Governor*

The parameter *powersave_bias* is important in *Ondemand*. Its main effect is to reduce energy consumption by reducing system performance. Specifically, it reduces the upper limit of frequency selection and compresses it in equal proportion. The range of value of *powersave_bias* is 0 – 1000. The value of 100 means reducing the original frequency by 10%.

We illustrate the effect of *powersave_bias* by running the two benchmarks on the laptop, setting the *powersave_bias* to three different values: 0, 200 and 400. The results of the experiment are shown in Figure 5.

The frequency selection is not restricted when the *powersave_bias* is 0. When the *powersave_bias* is set to 200 and 400, the frequencies represented by the yellow lines are significantly lower and their upper boundary is set at 2.0 GHz and 1.7 GHz under the same benchmark. This mode of behaviour allows the CPU to be in a specific frequency band to save more energy. When the deadline is equal to the task period, we can see that none of the two benchmarks are timed out, and the frequency regulation trend is flatter than when the *powersave_bias* is set to 0, which saves some energy while still meeting performance requirements.

C. *Analysis of Conservative Governor*

The main difference between the *Conservative* governor and the *Ondemand* governor is the speed at which the frequency changes. The *Conservative* governor has two thresholds, one for *up_threshold* and the other for *down_threshold*. The *Ondemand* governor will immediately set the frequency to the maximum frequency when the load is greater than *up_threshold*. *Conservative*, on the other hand, differs from this in that it adds a value which is *freq_step* to the current frequency when the load is greater than *up_threshold*, thus has a slow change in frequency. Similarly, it reduces the frequency by a value when the load is less than *down_threshold*. The *Conservative* governor has no *powersave_bias* value and is otherwise similar to *Ondemand* governor.

We did the same experiment on the *Conservative* as *Ondemand* to visualize its policy. The results of the experiment are shown in Figure 6. For the *FaceAndAudioRecog* benchmark, a clear step-up and step-down can be seen on all three platforms. It can be seen that in the middle part of each period, when the first half of the benchmark ends running, instead of immediately setting the frequency to the lowest frequency, the *Conservative* drops the frequency by one step for each time interval. Also, When the second half of the benchmark starts, the frequency rises in small steps. This step-down is more evident in *Basicmath*.

Comparing the experiments of *Ondemand* and *Conservative* shows that the advantage of *Conservative* over *Ondemand* is that it is more power efficient for benchmarks that do not require much performance. For benchmarks that do not require much performance, the *Conservative* governor saves more energy as it slowly increases the frequency. On the laptop platform, the *FaceAndAudioRecog* finished the first half

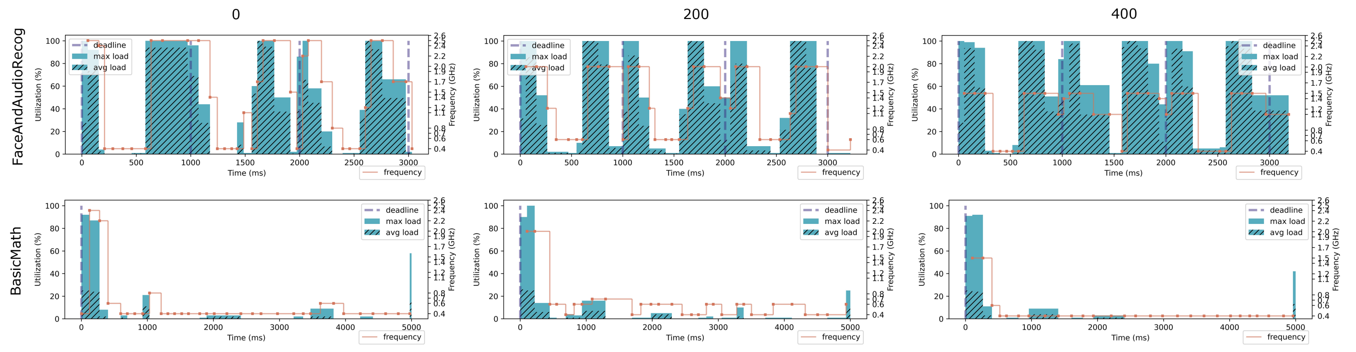
Fig. 5. The performance of two benchmarks on Laptop under *Ondemand* governor with tuning *powersave_bias*.

.

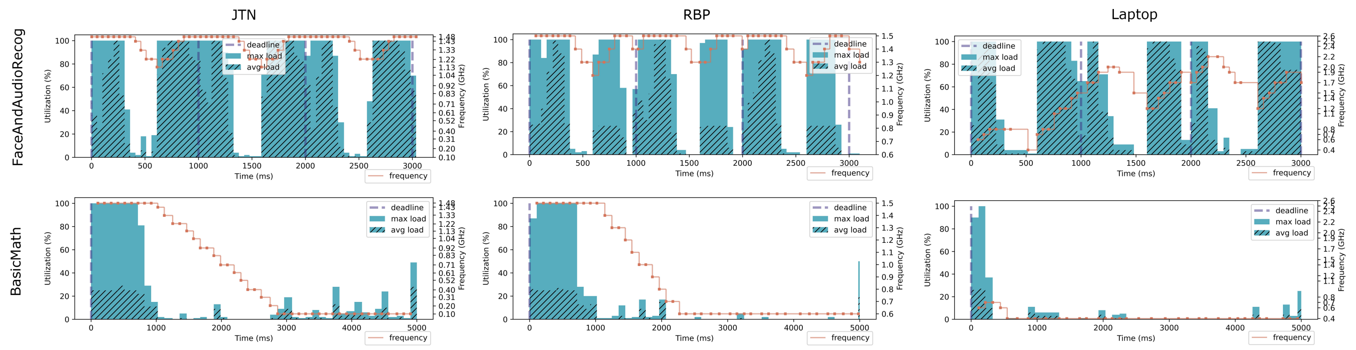
Fig. 6. The performance of *FaceAndAudioRecog* (the top row) and *Basicmath* (the bottom row) under *Conservative* governor on three platforms.

running without even rising to the maximum frequency. In this case, the *Ondemand* immediately adjusts the frequency to keep it at its maximum. In fact, this benchmark does not require such high performance with the set deadline. Thus, *Ondemand* wastes energy compared to *Conservative*.

On the other hand, the slow drop-off frequency of *Conservative* at the end of the benchmark wastes energy compared to the immediate drop rate of the *Ondemand*.

The *Conservative* is therefore good for tasks that do not require high performance, and its slowly increasing frequency meets the need for energy efficiency. However, for some large tasks, the slow increase in frequency can lead to slow start-ups that do not meet performance requirements. Overall, *Conservative* is slightly more energy efficient than *Ondemand*.

IV. Challenges in Practical Hardware

Integrating the DVFS algorithms into actual hardware and evaluating their energy consumption in actual hardware can often be difficult, which is quite different from the evaluation based on empirical formulae and experiments carried out in simulators. Through our experiments, we have summarised a few challenges that would be encountered in actual hardware: coarse-grained frequency levels and the OS interface not reflecting the actual frequency.

A. Coarse-Grained Voltage/Frequency Levels

As shown in Figure 1 and discussed in Section II, the two embedded devices have only two voltage levels. A constant voltage leads to a constant static power consumption, which means that adjusting only the frequency without adjusting the voltage leads to unsatisfactory energy savings. Only having two levels of voltage and frequency that DVFS policies can choose leads to a limitation of the policy choices, which is a challenge to developing a good DVFS policy.

B. Frequency Reflected by the OS Interface

When verifying the relationship between frequency and energy consumption, it is important to make sure that the CPU frequency set at the operating system level is reflected in the hardware. We have found that this is not always true and cannot be easily verified from the interfaces provided by Linux. In this part, we will discuss this issue and present a simple and easy-to-implement method for frequency verification.

The experiment is as follows. For each of the frequencies shown on the *scaling_available_frequencies* interface,

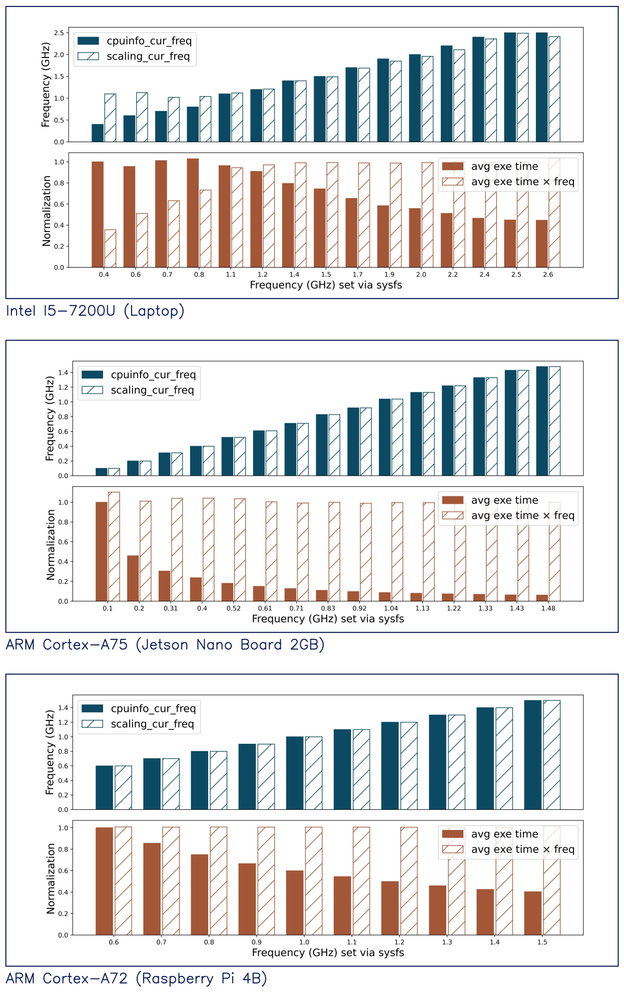

Fig. 7. Validate the frequency supported on three platforms.

run a CPU-intensive benchmark multiple times. This CPU-intensive benchmark consists of a long loop to do multiplication calculations. Under each frequency, we read and record the frequency from the two interfaces, *cpuinfo_cur_freq* and *scaling_cur_freq*, and get the average execution time of the benchmark. The frequency to read from these two interfaces is not high and does not put undue strain on the system. The result of this experiment is shown in Figure 7. For our experiments on all three platforms, the devices were connected to the power only and runs only a single task.

As can be seen in Figure 7, the two interfaces do not show the same frequency on the laptop platform. We verify which of the displayed frequencies are really set in the hardware using the following Equation 8, where f is the frequency, T is the execution time and C is the constant. This equation holds when the task is constant and no other external devices consume power.

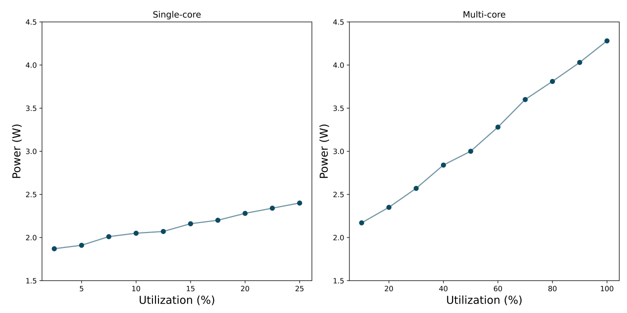

Fig. 8. The relationship of utilization and power of running a single-core benchmark (left) and multi-core benchmark (right).

$$f \times T \approx C \tag{8}$$

The red diagonal bars in the figure show the product of time and frequency. On the laptop, the *scaling_cur_freq* interface reflects the true frequency supported by the system, which is in the range of 1.4 GHz to 2.5 GHz. On embedded platforms, the two interfaces, *cpuinfo_cur_freq* and *scaling_cur_freq* interface, have relatively consistent values at each frequency. Based on the relationship between frequency and execution, we found that 0.1 GHz to 1.48 GHz is supported by the system on the JTN platform, and 0.6 GHz to 1.5 GHz is supported by the system on the RPB platform.

V. Utilization and Power

In this section, we will illustrate how utilization affects power at a given frequency. We perform the following experiments to illustrate the relationship.

We run a CPU-intensive benchmark and control the value of frequency and utilization for each run. The power is calculated by the measured energy consumption and time spent. The way we control the utilization is to control CPU running time and idle time. For example, we could let the benchmark run once and then let the system sleep the same amount of time as the run time of the benchmark to get a utilization close to 50%.

First, we run the single-core CPU-intensive benchmark, which means the maximum utilization of the four cores is 25%. The frequency is set to 1.224 GHz. The results are shown in Figure 8 (left). It can be seen that as utilization increases, the power is also increased. Similar trend can be observed for other frequencies. We then replace the single-core benchmark with a multi-core benchmark which means the maximum utilization can reach 100%. The multi-core benchmark runs four threads of the single-core benchmark at the same time. The results are shown in Figure 8 (right). The power consumption of the multi-core benchmark and that of the single-core benchmark show a similar trend.

VI. Conclusion

We have designed a set of experiments to illustrate DVFS power management on real devices operating under Linux

OS. The devices include an Intel-based laptop, an ARM-based Jetson Nano Board, and a Raspberry Pi platform. Our first set of experiments illustrates DVFS power model and provides an understanding of dynamic and static power related to frequency and voltage. Our second set of experiments visualizes the Linux governors, *Ondemand* and *Conservative*, through profiling OS kernels. The profile shows the real-time sequence of CPU max load, average load and frequency selected by the governors. The visualization helps the DVFS designer to better understand how the high-level DVFS software controllers are reflected at the hardware level. Furthermore, we have experimentally identified two aspects of DVFS in real hardware that need attention: the hardware's coarse voltage/frequency levels and the OS interface possibly not reflecting the actual frequency. These are issues that are most likely to be overlooked in experiments but are nevertheless important. Finally, we explored the relationship between utilization and power presented on a real board.

In this work, the link between frequency, utilization and static and dynamic power consumption is explored through experiments on real devices, helping researchers to better understand the parameters that affect power consumption and DVFS affects OS performance and power. At the same time, the experiments are designed to be easily replicated by researchers to study the performance of power consumption on other devices. Overall, The practical experience and experimental design shown in this work will bridge the gap in software DVFS design and its deployment into hardware devices.

References

[1] J.-G. Park, N. Dutt, and S.-S. Lim, "An interpretable machine learning model enhanced integrated cpu-gpu dvfs governor," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 6, pp. 1–28, 2021.

[2] J. L. C. Hoffmann and A. A. Fröhlich, "Online machine learning for energy-aware multicore real-time embedded systems," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 493–505, 2021.

[3] S. Bateni and C. Liu, "Apnet: Approximation-aware real-time neural network," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 67–79.

[4] R. Medina and L. Cucu-Grosjean, "Work-in-progress: Probabilistic system-wide dvfs for real-time embedded systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 508–511.

[5] Z. Dong and C. Liu, "Work-in-progress: New analysis techniques for supporting hard real-time sporadic dag task systems on multiprocessors," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 151–154.

[6] G. Massari, F. Terraneo, M. Zanella, and D. Zoni, "Towards fine-grained dvfs in embedded multi-core cpus," in *Architecture of Computing Systems – ARCS 2018*, M. Berekovic, R. Buchty, H. Hamann, D. Koch, and T. Pionteck, Eds. Cham: Springer International Publishing, 2018, pp. 239–251.

[7] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, 2019, pp. 1–8.

[8] C. Scordino, L. Abeni, and J. Lelli, "Real-time and energy efficiency in linux: Theory and practice," *SIGAPP Appl. Comput. Rev.*, vol. 18, no. 4, p. 18–30, jan 2019. [Online]. Available: https://doi.org/10.1145/3307624.3307627

[9] F. Reghenzani, A. Bhuiyan, W. Fornaciari, and Z. Guo, "A multi-level dpm approach for real-time dag tasks in heterogeneous processors," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 14–26.

[10] A. Alsheikhy, S. Han, and R. Ammar, "Delay and power consumption estimation in embedded systems using hierarchical performance modeling," in *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2015, pp. 34–39.

[11] L. Han, L.-C. Canon, J. Liu, Y. Robert, and F. Vivien, "Improved energy-aware strategies for periodic real-time tasks under reliability constraints," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 17–29.

[12] Y. Cho, D. Shin, J. Park, and C.-G. Lee, "Conditionally optimal parallelization of real-time dag tasks for global edf," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 188–200.

[13] D. Ramegowda and M. Lin, "Energy efficient mixed task handling on real-time embedded systems using freertos," *Journal of Systems Architecture*, vol. 131, p. 102708, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122002016

[14] S. K. Panda, M. Lin, and T. Zhou, "Energy efficient computation offloading with dvfs using deep reinforcement learning for time-critical iot applications in edge computing," *IEEE Internet of Things Journal*, pp. 1–1, 2022.

[15] T. Zhou and M. Lin, "Deadline-aware deep-recurrent-q-network governor for smart energy saving," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 6, pp. 3886–3895, 2022.

[16] A. Castagnetti, C. Belleudy, S. Bilavarn, and M. Auguin, "Power consumption modeling for dvfs exploitation," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 2010, pp. 579–586.

[17] K. R. Stokke, H. K. Stensland, P. Halvorsen, and C. Griwodz, "High-precision power modelling of the tegra k1 variable smp processor architecture," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE, 2016, pp. 193–200.

[18] J. L. March, S. Petit, J. Sahuquillo, H. Hassan, and J. Duato, "Dynamic wcet estimation for real-time multicore embedded systems supporting dvfs," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*. IEEE, 2014, pp. 27–33.

[19] P.-C. Hsiu, P.-H. Tseng, W.-M. Chen, C.-C. Pan, and T.-W. Kuo, "User-centric scheduling and governing on mobile devices with big. little processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, pp. 1–22, 2016.

[20] C. Scordino, L. Abeni, and J. Lelli, "Energy-aware real-time scheduling in the linux kernel," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 601–608.

[21] A. Balsini, T. Cucinotta, L. Abeni, J. Fernandes, P. Burk, P. Bellasi, and M. Rasmussen, "Energy-efficient low-latency audio on android," *Journal of Systems and Software*, vol. 152, pp. 182–195, 2019.

[22] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd ed. Pearson Education, 2004.

[23] H. Veendrick, "Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 468–473, 1984.

[24] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 2010, pp. 189–194.

[25] R. Wysocki, "Cpu performance scaling," https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html, 2018.

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.