Journal of Computer Science and Cybernetics, V.38, N.4 (2022), 311–326
DOI no. 10.15625/1813-9663/38/4/16873

AN IN-DEPTH EVALUATION OF FREQUENCY-AWARE SCHEDULER FOR IMPROVING USER EXPERIENCE ON MOBILE DEVICES

GIANG SON TRAN[1,*], AXEL CARLIER[2], DANIEL HAGIMONT[2]

[1]*ICTLab, University of Science and Technology of Hanoi, Vietnam Academy of Science and Technology, Ha Noi, Viet Nam*
[2]*University of Toulouse, 2 rue Charles Camichel B.P. 7122, 31071 Toulouse Cedex 7, France*

Abstract. Mobile devices are more and more invading our daily life. Users of such devices expect to have a good experience, mainly linked with performance. However, higher performance also means a reduction in battery life, negatively contributing to the overall user experience. A common way to balance this performance-battery trade-off is to reduce CPU frequency when underload with Dynamic Voltage and Frequency Scaling. In a previous work, we introduced a Frequency Aware Completely Fair Scheduler (called FA-CFS), which helps reduce battery consumption and increase the smoothness of mobile interface browsing. However, the current evaluation of FA-CFS model is only at quantitative results of power consumption rather than user experience on using their mobile device. In this paper, we perform an in-depth evaluation of the FA-CFS model, both quantitative results for system performance evaluation and qualitative results for user experience on mobile device usage. The experiments show that FA-CFS can reduce the rate of interface frame time peaks by up to 40% in terms of quantitative results and obtains a quantifiable impact on the quality of user experience with a quicker, more responsive interface.

Keywords. User experience; Mobile systems; Process scheduler; Dynamic frequency.

1. INTRODUCTION

Computing technology has advanced so rapidly that mobile devices are naturally considered essential to every person. Every daily activity can be performed anytime, anywhere, with mobile devices, such as surfing the web or organizing events. Due to this convenience, people tend to use their mobile devices instead of their computers. Modern mobile CPUs, originally started as embedded processors, now evolve to near desktop-class multi-core processors [1], usually included with other components like modems, graphic and audio processors in a System-on-Chip model. This architecture aims to meet the increasingly high demand of users.

On the other hand, users' energy consumption is getting more concern when using mobile devices. Mobile devices' battery autonomy is an essential criterion when customers choose phones to buy [2]. Due to this critical role of energy consumption, operating systems running

*Corresponding author.
E-mail addresses: tran-giang.son@usth.edu.vn (G.S. Tran); axel.carlier@enseeiht.fr (A. Carlier); daniel.hagimont@enseeiht.fr (D. Hagimont).

on mobile devices need to minimize power consumption. One popular method is dynamically adjusting the CPU frequency and voltage at runtime. This mechanism is called Dynamic Voltage and Frequency Scaling (DVFS). A CPU governor [3] in the operating system kernel is responsible for managing DVFS based on the required workload: it increases CPU frequency when the workload is high to meet this demand and vice versa.

Currently, the governor does not cooperate with the scheduler, a component in the kernel to orchestrate tasks running simultaneously. The default scheduler in Linux, Completely Fair Scheduler (CFS), uses time slice estimation to select running tasks. In our previous work [4], we proposed an extended version of CFS, called Frequency-Aware Completely Fair Scheduler (FA-CFS), which rebalances the workload time slice according to the differences in frequency. Through the experiments, we showed that the FA-CFS model helps reduce power consumption and increase the smoothness when using mobile devices. However, our previous evaluation of the FA-CFS model was only performed to assess quantitative results on power consumption, and we did not consider the qualitative results of user experience when using the enhanced FA-CFS scheduler on mobile devices.

In this paper, to tackle the missing evaluation on user experience, we perform an in-depth evaluation of FA-CFS in terms of (1) quantitative results for system performance evaluation and (2) qualitative results for user experience when using mobile devices with FA-CFS. Our contributions to this evaluation are three folds:

- More experiments are performed on more mid-range mobile devices to evaluate the effectiveness of FA-CFS for reducing power consumption and increasing the smoothness of using mobile devices.
- More analyses are provided to clarify the helpfulness of FA-CFS for reducing power consumption and increasing the smoothness of using mobile devices.
- New experiment scenarios are performed to evaluate the qualitative results of user experience on mobile devices running with FA-CFS.

The remainder of this paper is organized as follows. Section 2 discusses related works regarding energy-aware schedulers and CPU allocation. Section 3 introduces the principle and algorithm of the frequency-aware scheduler. We detail our experiments with an in-depth analysis of quantitative results in Section 4 and qualitative results in Section 5. Section 6 concludes our paper and presents possible perspectives.

2. RELATED WORKS

CPU is among the most power-hungry components in any device, including smartphones. Many research works have been dedicated to analyzing power consumption and optimizing the operating system scheduler to increase the time between battery charges.

A detailed power consumption analysis of different mobile phone components is evaluated and presented by Carroll *et al.* in [5]. The authors developed an energy model for a mobile device and measured power breakdown in various states in this work. In suspended and idle states, the CPU consumption is the 2^{nd} and the 3^{rd} most power-hungry component, respectively, below the GSM module and screen. Similarly, Nguyen *et al.* [6] categorize different offloading-based approaches to reduce CPU power consumption in a mobile phone.

This method is essential in minimizing power consumption since a mobile CPU is much slower when compared with a desktop- or server-class counterpart. However, this approach needs to balance the trade-off between reduced CPU energy with increased data transmission power.

Using DVFS to reduce CPU consumption is another popular operating system-level approach. Annamalai *et al.* [7] proposed a method to reduce energy per instruction (or increase energy efficiency) for asymmetric multi-core systems using dynamic resource allocation and DVFS. Asymmetric multi-processor is widely used in many modern mobile CPU architectures. The authors achieved 17.9% energy per instruction reduction compared to the baseline heterogeneous multi-core system. Another work by Zhu *et al.* [8] proposed non-work-conserving scheduling to exploit discounted energy consumption in mobile phones, especially those with CPUs of various micro-architectures. This work claims that scheduling background tasks to low-utilized cores when these cores are enabled will benefit energy efficiency. The evaluation shows that this method can reduce up to 63% of energy with a negligible performance impact.

At the software level, several works attempted to provide energy optimization methods at compilation and runtime. Choi *et al.* [9] proposed an energy-aware framework by adapting Dynamic Power Management and DVFS policies with the required workload. This work achieved 22% to 79% power reduction while insignificantly affecting the workloads. In their recent work [10], Sheng *et al.* proposed an optimization method at the compilation stage for heterogeneous multi-core systems, which is important since almost every modern mobile phone has multiple core CPUs. This work extends the C language to support the source-to-source compiler, enabling source code optimization techniques for multi-core systems.

To optimize user experience, Kumar *et al.* [11] summarize different approaches for balancing the trade-off between energy and performance on a smartphone. The authors claim that energy response, performance, and dynamic user behavior are still open problems in the research domain. Little work has been done to level energy reduction and user experience, therefore it is inadequate to compare combination work with other methods. DVFS techniques can also be applied for this balance, as in our previous work [4].

3. METHODOLOGY

In this section, based on CFS, we present the FA-CFS model which we proposed in our previous work [4] in order to improve performance of mobile system.

3.1. CFS Model

Completely Fair Scheduler (CFS) is the default scheduler in Linux used to calculate time slices for running tasks in multi-core mobile systems. Firstly, we present the principle of CFS scheduling model and its performance penalties when taking into account core frequency changes.

Let W be a workload executed in a single thread and can be considered several CPU cycles required to perform a task. A workload is measured as a multiplication of speed and time. In the simplest case, if this workload is scheduled on a single-core CPU with constant frequency f (approximately [a] proportional to the number of instructions per second), we have

$$W = f \times T, \tag{1}$$

where T is the total time (in seconds) of execution.

In the default Linux kernel, the governor can adjust the frequency on one core after every sampling time τ_i (itself composed of CFS time slices).

Within each τ_i, the scheduler spends a little CPU time (ζ_i) for accounting and selecting the next scheduled thread after each time slice. This time can be considered as the performance overhead of the process scheduler. Therefore, T in equation (1) becomes

$$T = \sum_{i=1}^{n} (\tau_i + \zeta_i), \tag{2}$$

where n is the total number of sampling times during the execution duration.

The CPU frequency f is calculated by the governor at runtime based on the total workload of the core and is different for each τ_i. With the consideration of the overhead of T in equation (2) and the fluctuation of f in each time slice, W in equation (1) becomes

$$W = \sum_{i=1}^{n} f_i \times (\tau_i + \zeta_i), \tag{3}$$

where $f_{min} \leq f_i \leq f_{max}$ is the CPU frequency at sampling time τ_i.

In the Linux kernel, the governor's sampling time is calculated as a multiple of the CFS scheduler's time slice S_i: $\tau_i = \pi \times S_i$, where S_i is dynamically estimated at runtime by CFS model. Thus, we have

$$W = \sum_{i=1}^{n} f_i \times (\pi \times S_i + \zeta_i). \tag{4}$$

Taken into consideration that mobile processors are multi-core with N cores ($N > 1$), the running thread of the workload W can be migrated from the currently scheduled core j^{th} (having frequency f_i^j) to the actual working core k^{th} (having frequency f_i^k), and $i, k \in [0, N-1]$. In CFS model, a thread migration with the frequency change does not change the value of the time slice S_i, but cause a performance penalty δ_i, which is the difference between the workload at the current core j^{th} (calculated as $f_i^j \times (\pi \times S_i + \zeta_i^j)$) and the workload at the actual working core k^{th} (calculated as $f_i^k \times (\pi \times S_i + \zeta_i^k)$). Consequently, the formula to compute the performance penalty δ_i is as follows

$$\delta_i = (f_i^j - f_i^k) \times \pi \times S_i + f_i^j \times \zeta_i^j - f_i^k \times \zeta_i^k. \tag{5}$$

Since the amount of work (frequency $\times$ time) for accounting and scheduling can be considered as a constant between sampling intervals in CFS [12], we have $f_i^j \times \zeta_i^j \approx f_i^k \times \zeta_i^k$. As a result, equation (5) can be simplified as

$$\delta_i = (f_i^j - f_i^k) \times \pi \times S_i. \tag{6}$$

[a] All the equations below are theoretical and approximation of the reality.

In the workload duration, the total performance penalty Δ (in terms of the amount of work) is defined as

$$\Delta = \sum_{i=1}^{n} \delta_i = \sum_{i=1}^{n} ((f_i^j - f_i^k) \times \pi \times S_i). \tag{7}$$

Since the total penalty in equation (7) affects performance of the mobile system, it is important to reduce it as much as possible. In the next section, we will present an extension of CFS model, called Frequency-Aware CFS (FA-CFS), which we proposes in our previous work [4] in order to minimize this penalty.

3.2. FA-CFS Model

In our FA-CFS model, we introduce a new method to calculate the time slice S_i when a thread migration with the frequency change happens. Unlike CFS keeping the value of S_i unchanged in a migration between frequency changes, our method modifies the value of the time slice S_i to $S_i^{'}$ for the actual working core frequency. When applying this modification into the equation (6), the performance penalty δ_i becomes

$$\delta_i = f_i^j \times \pi \times S_i - f_i^k \times \pi \times S_i^{'}. \tag{8}$$

Since we want to minimize the total performance penalty, we compute the value of $S_i^{'}$ such that $\delta_i = 0$, therefore

$$f_i^j \times \pi \times S_i - f_i^k \times \pi \times S_i^{'} = 0. \tag{9}$$

Sovling the equation (9), we have

$$S_i^{'} = \frac{f_i^j}{f_i^k} \times S_i. \tag{10}$$

Using the equation (10) to calculate the time slice at the actual working core when a thread migration with the frequency change occurs, our FA-CFS model minimizes the performance penalty δ_i and therefore minimizes the total performance penalty Δ in equation (7).

3.3. Integration inside Android / Linux Kernel

We implemented our proposed frequency-aware scheduler in the Linux environment, where our model acts as a frequency-aware extension to the CFS scheduler (see Figure 1). We use the *CPUFreq* interface to call the governor for collecting CPU frequencies. The CPUFreq interface exports kernel-wide functions to provide CPU frequency knowledge to other components inside the kernel. Notable functions include:

- `cpufreq_get` provides the current frequency of a specific CPU / core.
- `cpufreq_get_max` provides the maximum frequency of a specific CPU / core.
- `cpufreq_get_policy` returns the current policy (containing the governor's name with a pair of configured minimum/maximum frequencies).

Having extracted frequencies, we implement our proposed algorithm, which adjusts weights and time slices in Linux's CFS according to the core frequencies.

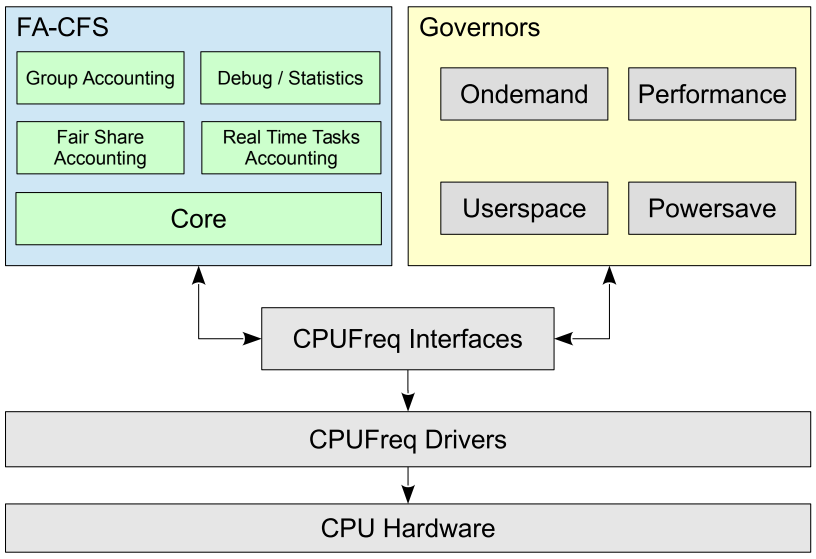

Figure 1: Implementation inside Linux Kernel.

4. SYSTEM PERFORMANCE EVALUATION

This section presents the responsiveness improvement for user interaction provided by our FA-CFS scheduler. We show this enhancement by comparing FA-CFS with the original CFS scheduler. We first describe the experiment's setting, its results, and then our analysis.

4.1. Experimental setup

Interface frame time measurement

The principal metric to evaluate our improvements in FA-CFS is *interface frame time.* We choose to measure frame time since it is essential in providing responsiveness for the user interface. A fully rendered frame is passed through a set of steps in Android rendering pipelines: *execute* the issued layout commands (t_e), *process* the swapping buffers (t_p), *prepare* the texture and finally (t_t) *draw* the content to the screen (t_d). Time for rendering each user interface frame t_f is accumulated from each of the above times

$$t_f = t_e + t_p + t_t + t_d. \tag{11}$$

Evaluation scenario

Our evaluation considers a popular scenario where users browse an online news website using smartphones and tablets. In the experiment, we choose `http://bbc.com` as the target website due to its popularity. Since we want to compare the efficiency of our FA-CFS with CFS, we ask all users to browse this news website sequentially using both of the schedulers. We measure the interface frame times generated by user interactions during browsing sessions in both steps.

A browsing session in our experiment includes: (1) the user starts the stock browser, (2) he types the URL `http://bbc.com`, (3) he waits for page load, (4) he scrolls up and down as soon as one or more parts of the page content appears. The user can click on any link on the page to navigate to another page (of the same BBC website) and repeat the last two steps (3) and (4). Each user session is limited to 100 seconds. In this scenario, there are three different workload types, generated by the UI thread (rendering and handling interaction), background network threads (to fetch data from the remote server), and the browser engine (in charge of parsing HTML and processing JavaScript with Chromium's V8 JavaScript engine). We clear the browser cache before starting each experiment session to avoid preloaded images.

Technical choices

Table 1: Hardware specifications of devices

	Mobile Phone		Tablet	
	LG Nexus 4	Xiaomi Redmi Note 4X	Asus Nexus 7 Wifi	Samsung Galaxy Tab A6 10.1
Chipset	Qualcomm Snapdragon S4 Pro	Qualcomm Snapdragon 625	Nvidia Tegra 3	Samsung Exynos 7870
CPU	Quad-core 1.5 GHz Krait	Octa-core 2.0 GHz Cortex-A53	Quad-core 1.2 GHz Cortex-A9	Octa-core 1.6 GHz Cortex-A53
GPU	Adreno 320	Adreno 506	ULP GeForce	Mali-T830 MP2
Memory	2GB RAM	3GB RAM	1GB RAM	2GB RAM
Screen	720x1280 4.7"	1080x1920 5.5"	800x1280 7"	1200x1920 10.1"

On the hardware side, our experiments are performed on two categories of mid-range Android devices: mobile phones (LG Nexus 4 and Xiaomi Redmi Note 4X) and tablets (Asus Nexus 7 Wifi and Samsung Galaxy Tab A6) (see Table 1). We would like to investigate the impact of our scheduler's improvement on different types of devices.

On the software side, we build from source an aftermarket open-source operating system called LineageOS, based on the Android Open Source Project (AOSP). We decided to build LineageOS from its source because of the ability to customize the Linux kernel and *flash* (or install) the kernel and the whole operating system into our devices. We use the last LineageOS version supported by all devices to implement our model. The Linux kernel version included in LineageOS for each device is as follows:

- LG Nexus 4: 3.14.0
- Xiaomi Redmi Note 4X: 3.18.24
- Asus Nexus 7 Wifi: 3.4.113
- Samsung Galaxy Tab A6 10.1: 3.18.14

We use an Android's developer option called "Profile GPU rendering" to monitor and gather interface frame times during the experiments. We then use Android's integrated "dumpsys" tool on the devices to collect statistical information with a USB cable, including the monitored interface frame time components (*execute*, *process*, *prepare* and *draw*).

4.2. Experimental results

Interface frame time peaks

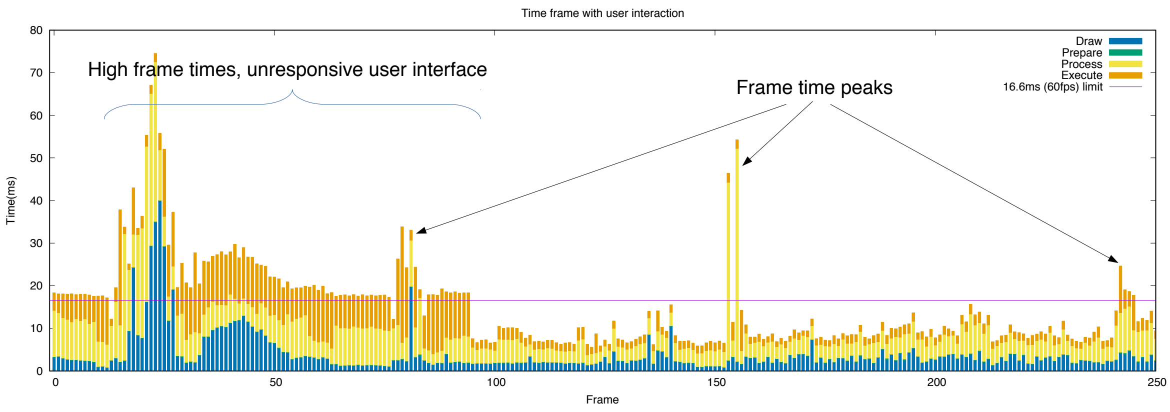

Figure 2: Interface frame time peaks on the Galaxy Tab A6 with CFS scheduler and Interactive governor.

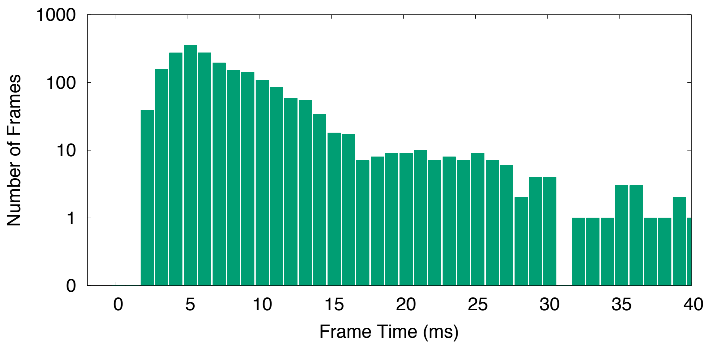

Figure 3: Frame time distribution of CFS on Samsung Galaxy Tab A6

Figure 2 shows a set of captured frame times extracted from one user session on the Galaxy Tab A6 using CFS and *interactive* governor. It can be depicted from this figure that frame times during this session are not stabilized but generally are smaller than the optimal 16.6ms. In the first part of this session (frame 0 - 90), frame times were relatively high because the web browser needs to perform three tasks at the same time: fetching web content from remote servers, parsing partial contents (HTML, CSS) as they arrive, executing JavaScript and rendering them (text with fonts, decoding and displaying images) to the screen. The rendering thread is not provided with enough computational power because the background threads overload the CPU. Thus, the UI thread struggles to maintain an optimal frame time. Since frame 100, page fetching and HTML parsing tasks have finished, but very high frame times exist. Some even exceeded 40ms (around frame 150-155).

These peaks (or spikes) cause "micro stuttering," a term that indicates irregular delays between rendered frames [13]. Micro stuttering decreases the responsiveness of the user interface, even though the average frame rate is high enough. These high frame time peaks can be explained as a consequence of CPU core frequency changes or thread migration among cores (with different frequencies).

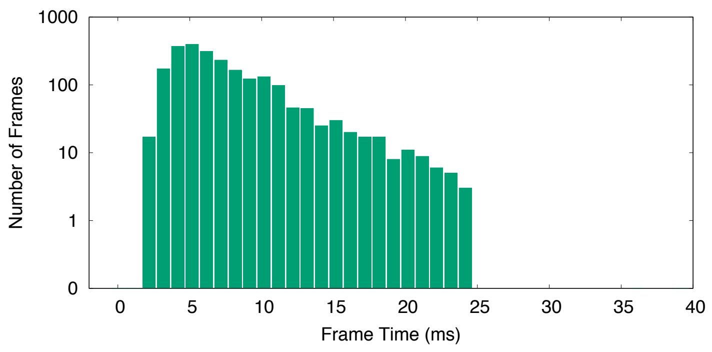

Figure 4: Frame time distribution of FA-CFS on Samsung Galaxy Tab A6

Frame time distribution

Due to the fine-grain nature of frame time (in milliseconds), we use a more statistical metric *frame time distribution* in the second step of our evaluation. We set up our experiment on a user session with the Samsung Galaxy Tab A6. During user interactions, we gather approximately 2200 frame times (around 90 seconds of browsing BBC website) of CFS and FA-CFS with the *interactive* governor, representing them as histograms in Figures 3 and 4, respectively.

It can be depicted from Figure 3 that CFS causes micro stuttering with frames longer than 16.6ms. Some frames take even more than 38ms (21.4ms longer than the 16.6ms limit). These peaks cause choppy web content scrolling in the browser. Applying our FA-CFS into LineageOS significantly reduces these peaks (Figure 4). The maximum frame time for FA-CFS is 24ms compared to 40ms on CFS. In the session, FA-CFS produces only 42 frames longer than the 16.6ms limit. In contrast, this number of the CFS counterpart is 97. From these results, our FA-CFS achieves a reduction of 40% frame time peaks (97 frames down to 47 frames). Additionally, frame times are better packed in the mean 5-6ms range. These two figures clearly illustrate the benefit of improving the user experience of our FA-CFS.

Top peak ratio

The previously described two metrics, frame time and frame time distribution, are not adequate to measure web-browsing sessions in a reliable way on a larger scale because many factors can affect frame time in such a session: delay of network connection, dynamic content of the webpage (e.g., advertisement) or even the workload of the server.

To analyze the effectiveness of our FA-CFS scheduler, we define a statistical metric called *top peak ratio* (TPR). The metric is described as follows: a top $x\%$ peak ratio at y milliseconds shows that during the experiment, the top $x\%$ of all frame times are longer than y milliseconds. TPR represents the stability of frame time and, thus, the "quality" of responsiveness in user interactions. In this part of the evaluation, we analyze the average TPR values of all user sessions.

A total of 5 users were involved in this part of our experiment. On each device, users performed three sessions with the CFS scheduler and three sessions with our FA-CFS scheduler, and each session uses a different governor. The three most popular governors with other characteristics were used to manage the rise and decline of system load with frequency ramp up and ramp down. The governors included in our experiments are *interactive* (default,

Table 2: Average TPR (ms) of CFS vs. FA-CFS with 3 governors on Nexus 7 Wifi

	Interactive			Ondemand			Performance		
TPR	CFS	FA-CFS	±	CFS	FA-CFS	±	CFS	FA-CFS	±
Max	48.58	37.3	-23.2%	55.12	44.05	-20.1%	31.21	30.85	-1.2%
1	38.94	29.12	-25.2%	41.77	35.10	-16.0%	23.68	25.23	6.5%
2	31.29	23.16	-26.0%	32.83	25.57	-22.1%	19.03	19.49	2.4%
3	27.25	17.21	-36.8%	29.83	21.29	-28.6%	16.79	**16.47**	-1.9%
4	23.13	**16.08**	-30.5%	23.49	16.73	-28.8%	**15.62**	15.88	1.7%
5	20.44	15.98	-21.8%	22.89	**16.25**	-29.0%	14.73	14.61	-0.8%
6	18.89	15.45	-18.2%	19.96	15.88	-20.4%	13.98	14.05	0.5%
7	18.22	14.93	-18.1%	18.72	15.65	-16.4%	13.29	13.42	1.0%
8	17.76	14.68	-17.3%	18.45	15.45	-16.3%	12.75	12.60	-1.2%
9	17.33	14.46	-16.6%	18.11	15.24	-15.8%	12.36	12.21	-1.2%
10	17.18	14.27	-16.9%	17.94	15.02	-16.3%	11.93	11.81	-1.0%

Table 3: Average TPR (ms) of CFS vs. FA-CFS with 3 governors on Galaxy Tab A6

	Interactive			Ondemand			Performance		
TPR	CFS	FA-CFS	±	CFS	FA-CFS	±	CFS	FA-CFS	±
Max	42.11	37.29	-11.4%	44.18	42.43	-4.0%	28.06	27.61	-1.6%
1	35.17	32.92	-6.4%	35.75	33.59	-6.0%	22.87	23.55	3.0%
2	31.87	30.73	-3.6%	34.07	31.28	-8.2%	21.67	22.94	5.9%
3	30.53	29.92	-2.0%	31.92	30.39	-4.8%	20.88	20.6	-1.3%
4	28.42	26.21	-7.8%	26.01	23.96	-7.9%	18.41	17.9	-2.8%
5	22.63	23.64	4.5%	21.76	19.23	-11.6%	**15.88**	**15.71**	-1.1%
6	20.73	21.26	2.6%	19.40	17.33	-10.7%	14.24	14.35	0.8%
7	18.23	**16.23**	-11.0%	17.48	**16.07**	-8.1%	13.26	13.48	1.7%
8	**16.44**	14.04	-14.6%	17.17	14.15	-17.6%	12.80	12.54	-2.0%
9	15.79	13.34	-15.5%	**16.19**	13.82	-14.6%	12.23	11.75	-3.9%
10	14.96	12.78	-14.6%	15.54	13.11	-15.6%	11.97	11.35	-5.2%

fastest ramp up with intermediate frequencies, and best latency), *ondemand* (short ramp-up, fast ramp down, mostly minimum, and maximum frequencies only), and *performance* (keep the highest frequency, waste energy). Before changing the governor, we reboot the device to avoid caching on CPUs. Therefore, each user is requested to perform a total of 30 browsing sessions (6 on each of the four Android devices).

Tables 2 - 5 show average top peak ratio of all user sessions on all devices with 3 different governors. It is worth reminding that *interactive* is the default governor on most mobile devices. Bold numbers indicate the minimum TPR at which FA-CFS can maintain frame time less than the 16.6ms limit.

Frame time reduction. With the two highly dynamic governors, *interactive* and *ondemand*, these tables generally show that FA-CFS achieves better frame times than CFS. The Nexus 7 benefits greatly from our time slice optimization, with an average of 16.9% and 16.3% reduction in time decreased (with *interactive* and *ondemand*, respectively) for TPR 10% (Table 2). While showing less improvement regarding top TPR on Nexus 4, FA-CFS still

Table 4: Average TPR (ms) of CFS vs. FA-CFS with 3 governors on Nexus 4

	Interactive			Ondemand			Performance		
TPR	CFS	FA-CFS	±	CFS	FA-CFS	±	CFS	FA-CFS	±
Max	35.36	30.31	-14.3%	36.65	33.41	-8.8%	24.98	25.21	0.9%
1	24.52	20.36	-17.0%	23.83	22.01	-7.6%	19.32	19.38	0.3%
2	18.24	17.14	-6.0%	19.27	19.41	0.7%	**16.42**	17.31	5.4%
3	17.58	**16.56**	-5.8%	18.04	16.75	-7.2%	15.16	**16.42**	8.3%
4	**16.49**	15.86	-3.8%	17.50	**15.83**	-9.5%	14.02	14.50	3.4%
5	16.03	15.42	-3.8%	16.91	15.29	-9.6%	13.34	14.06	5.4%
6	15.63	15.27	-2.3%	**16.49**	14.67	-11.0%	12.99	13.75	5.9%
7	15.48	14.81	-4.3%	16.19	14.32	-11.6%	12.51	13.28	6.2%
8	14.82	14.25	-3.8%	16.03	13.87	-13.5%	12.15	12.85	5.8%
9	14.78	14.18	-4.1%	15.62	13.45	-13.9%	11.86	12.53	5.6%
10	14.54	14.03	-3.5%	15.41	13.14	-14.7%	11.58	12.32	6.4%

Table 5: Average TPR (ms) of CFS vs. FA-CFS with 3 governors on Redmi Note 4

	Interactive			Ondemand			Performance		
TPR	CFS	FA-CFS	±	CFS	FA-CFS	±	CFS	FA-CFS	±
Max	33.64	29.21	-13.2%	30.11	28.61	-5.0%	28.40	26.24	-7.6%
1	28.31	25.72	-9.1%	25.48	23.25	-8.8%	23.86	22.56	-5.4%
2	21.24	17.21	-19.0%	22.48	22.58	0.4%	17.72	17.75	0.2%
3	18.50	**15.51**	-16.2%	20.53	19.09	-7.0%	**15.01**	**14.84**	-1.1%
4	**15.33**	13.70	-10.6%	19.47	**16.18**	-16.9%	13.90	13.95	0.4%
5	14.56	13.14	-9.8%	18.15	15.66	-13.7%	12.75	12.40	-2.7%
6	14.01	12.95	-7.6%	17.21	15.34	-10.9%	12.02	11.88	-1.2%
7	13.78	12.22	-11.3%	16.64	14.04	-15.6%	11.41	11.33	-0.7%
8	13.42	11.86	-11.6%	**15.67**	13.72	-12.4%	10.93	10.91	-0.2%
9	13.07	11.58	-11.4%	14.97	13.57	-9.4%	10.45	10.70	2.4%
10	12.79	11.25	-12.0%	14.22	13.35	-6.1%	10.17	10.53	3.5%

achieves 3.8% and 14.7% enhancement for TPR 10% (Table 4). The faster devices (Galaxy Tab A6 and Redmi Note 4) exhibit less improvement than the slower ones. It can be seen that the general TPR of the Galaxy Tab A6 and the Redmi Note 4 could achieve only around 14.6%/14.6% and 12%/6.1% for TPR 10% using FA-CFS with *interactive* and *ondemand*, respectively (Table 3 and 5). The differences between these devices can be interpreted as a difference in hardware configuration: double amount of CPU cores (octa-core vs. quad-core), faster CPU speed (2.0GHz/1.6GHz and 1.5GHz/1.2GHz), and a larger amount of memory.

Frame time stabilization. Not only does our frequency-aware FA-CFS scheduler reduce average frame times, but also it provides better frame time stabilization than traditional CFS: TPR 1%, 2%, 3%, and 4% provide significant improvements on both devices. Especially with a better TPR 1% peak ratio (25.2% and 16% reduction for *interactive* and *ondemand* on Nexus 7) (Table 2), user has smoother and more responsive interface as well as experiences less micro stuttering frames during their interactions.

The 16.6ms limit. FA-CFS can keep more frames shorter than the 16.6ms limit, or in other words, is better in terms of providing a *good user experience.* In most cases, FA-CFS can keep 1%-2% fewer over-the-limit frames than CFS. For instance, the Galaxy Tab A6 *interactive* (7% vs 8% peak frames) and *ondemand* (7% vs 9% peak frames) (Table 3). We can see the same trend on Nexus 4: *interactive* (3% vs 4% peak frames) and *ondemand* (4% vs 6% peak frames) (Table 4). In an extreme case, the Nexus 7, FA-CFS completely outperforms CFS using both dynamic governors: *interactive* (4% vs 10+% peak frames) and *ondemand* (5% vs 10+% peak frames) (Table 2).

Tablet vs. Phone. One important observation we learned during the experiment is that the frame time of mobile phones is generally shorter than tablets. For example, the Nexus 7 and Nexus 4 both have quad-core processors and similar screen resolution (800×1280 vs. 720×1280), but the Nexus 7 has a maximum frame time 37.4% longer (48.58ms vs. 35.36ms) with *interactive* and 65% longer (55.12ms vs. 33.41ms) with *ondemand* than Nexus 4 using CFS scheduler. The same difference can be observed even with faster devices: Galaxy Tab A6 has a maximum frame time 25% longer (42.11ms vs. 33.64ms) with *interactive* and 20% longer (44.18ms vs. 36.65ms) with *ondemand* than Redmi Note 4 using CFS scheduler, although having almost the same resolution (1200×1920 vs. 1080×1920) and octa-core CPUs. This variation can be explained as although having approximately the same number of pixels, the tablets have to draw more elements to the screen than the phones (e.g., more text elements and more images) due to a physically larger screen size. The size difference results in more layout operations, scaling operations, and rasterization operations in the rendering process.

Performance governor. Last but not least, the right parts of these tables exhibit almost no improvement for all devices using *performance* governors with all user sessions. This phenomenon can be explained that *performance* always provides maximum CPU computational power to all possible threads without frequency changes ($\frac{f_i^j}{f_i^k} = 1$).

The analyses of Tables 2 - 5 above show that our FA-CFS enhances frame time stabilization, increases average frame rate, and reduces frame time peaks (or spikes) with widely used governors (*interactive* and *ondemand*). Due to this, our FA-CFS scheduler proves its efficiency in improving user experiences while interacting with mobile devices.

5. QUALITY OF EXPERIENCE EVALUATION

In this section, we present the user study we performed to evaluate the impact of our frequency-aware scheduler on the Quality of Experience (QoE).

5.1. Protocol

Our goal in this user study is to provide evidence that user experience is affected by the change in the scheduler. We ask users to browse several popular websites on different devices and under various scheduling conditions. We explain to users that two different modes (i.e., schedulers, but we do not enter into details) will be used during the experiment, and between them, one is perceptibly better. We then start the study.

Tasks. The study is made of a succession of five similar tasks for each user. Each task consists of presenting the user a given device (randomly chosen between a phone and a tablet), requesting them to browse (mostly scroll up and down) a given web page using both schedulers (in randomized order) until it is fully loaded, and asking them to choose the scheduler with the highest perceived QoE. All users are requested to repeat the task five times, each for a different website in a shuffled sequence. Randomization is used to prevent any ordering bias that could occur during the study. By the end of this study, each user should produce five data points, each for one described task on a given web page.

Devices. We used a Galaxy Tab A6 and a Redmi Note 4 in our experiments. The goal is to assess whether the change in the scheduler is perceptible both on tablets and smartphones.

Websites. We ask users to browse five popular websites that any user would typically browse at least once a week, namely a Facebook timeline (i.e., a social network), the BBC main page (i.e., a news website), a Wikipedia page (`https://en.wikipedia.org/wiki/United_States`, chosen to have a considerable enough loading time), an Amazon page (to represent e-commerce websites), and the Mac page on Apple's website (a company portal).

Questionnaire. We ask users questions after each of the ten similar tasks, and we ask a different set of questions at the end of the study. We ask two questions after each task:

- How perceivable, from 1 (no perceivable difference) to 5 (noticeable difference), is the difference in Quality of Experience between the two modes?
- Which mode (the first or the second) did you prefer in this task?

At the end of the study, we also ask a few more open questions to try to characterize the effects of the change in the scheduler:

- What made you perceive the difference between the two modes? ("I did not see any difference" is an acceptable answer)
- When was this difference the most obvious? (examples of possible answers indicated to users: at the beginning or the end of the page loading, after a particular interaction, etc.)
- Was the difference more obvious on one or the other of the two devices?
- Was the difference more evident on some of the websites?

Users. A total of forty-two users (thirty-eight male and four female) performed this task, with ages ranging from 19 to 22. The user target may seem very specific, but we were particularly interested in users who are very familiar with mobile devices and browse the Internet every day on their devices, which explains this choice. These users produced a total of 210 data points (perceived difference and preference per website).

5.2. Results

We assign the following scores to users' choice of preference

$$\text{preference} = \begin{cases} 1 & \text{if the user preferred mode 1,} \\ 0 & \text{if the user could not see any difference,} \\ -1 & \text{if the user preferred mode 2.} \end{cases} \tag{12}$$

We find the average user preference to be 0.69, with a 99% confidence interval ranging in [0.59, 0.78]. This result indicates a very perceivable difference between the two modes, in favor of mode 1 (our proposed solution), which can definitely not be attributed to chance. We also divided the users' answers according to the device they used. The average user preference on the phone setup is 0.71 (into [0.57, 0.85] with a 99 % confidence) whereas it falls to 0.66 (into [0.53, 0.79] with a 99 % confidence) on tablet setup.

We asked users to quantify on a 1-to-5 scale how perceivable the difference between the two modes was. We refactor the results from -4 (resp. 4), which denotes a significant difference in favor of mode 2 (resp. mode 1), to -1 which represents a slight difference in favor of mode 2 (resp. mode 1). 0 means there was no perceivable difference between the two modes. We observe an average perceived difference of 1.87, with a 99% confidence interval ranging into [1.78, 1.96]. In the perception scale we provided to users, this value corresponds to "medium difference."

Answering our open questions in the study, the users are able to distinguish the schedulers mostly because of the difference in scrolling performance (57% users) and in the total loading time (38% users). Majority of users indicate that this difference is most obvious during scrolling (43%), before images are shown (33%) and when they start scrolling (26%). This indicates the effectiveness of our contribution to reduce frame time peaks.

Finally, we also investigated the variation of users' preferences on the different websites, but we can not see any significant difference in our collected data.

All these results confirm that the quantitative results we presented in the previous section have a quantifiable impact on the users' Quality of Experience. Among the comments made by the users during the study, the most common one was that our method tends to lead to a quicker, more responsive interface.

6. CONCLUSION AND PERSPECTIVES

In mobile operating systems such as Android, the process scheduler does not consider dynamic processor frequency modifications, which reduce energy consumption. Consequently, tasks may be penalized when executed on a processor with a reduced frequency. This penalty is a critical issue for interactive tasks, which may lead to unresponsiveness of user interface to user interactions, thus reducing user experience on the mobile device. In this paper, we performed an in-depth evaluation of the effectiveness of our previously introduced FA-CFS model, not only for system performance but also for the quality of user experience when using their mobile device. Through the experiments, the results show that FA-CFS reduces unresponsiveness of user interface to user interactions (up to 40%) as well as noticeably perceived difference by the users, and therefore, enforces the smoothness of user interface.

This work opens several perspectives. First and foremost, since our work helps improve user experience on mobile systems, it is worth investigating our model in various situations to see if it can bring benefits: with different workloads on mobile devices or larger multi-core and multi-CPU platforms (i.e., desktops and virtualized servers). Secondly, combining our frequency-aware scheduler with a performance-oriented scheduler (e.g., BFS scheduler) is also an interesting research direction. Our FA-CFS scheduler can consider the BFS scheduler's advancements to improve UI responsiveness and save CPU power. Last but not least, we wonder if our frequency-aware scheduling policy can be better integrated into the Red-Black tree by restructuring it based on core frequencies at runtime.

ACKNOWLEDGEMENTS

The authors would like to thank Vietnam Academy of Science and Technology (VAST) for funding this research in the scope of project DL0000.05/20-22.

REFERENCES

[1] M. Halpern, Y. Zhu, and V. J. Reddi. "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)*, pages 64–76. IEEE, 2016.

[2] D. Ferreira, A. K. Dey, and V. Kostakos, "Understanding human-smartphone concerns: A study of battery life", in *Pervasive Computing. Pervasive 2011. Lecture Notes in Computer Science,* vol 6696. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/978-3-642-21726-5_2`

[3] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts essentials.* John Wiley & Sons, Inc., 2014.

[4] G. S. Tran, T. P. Nghiem, T. V. Ho, and C. M. Luong, "Extended process scheduler for improving user experience in multi-core mobile systems", in *SoICT '16: Proceedings of the 7th Symposium on Information and Communication Technology,* December 2016, pp. 417–424. `https://doi.org/10.1145/3011077.3011106`

[5] A. Carroll, G. Heiser, et al. "An analysis of power consumption in a smartphone" `https://www.usenix.org/legacy/event/atc10/tech/full_papers/Carroll.pdf`

[6] Q.-H. Nguyen and F. Dressler, "A smartphone perspective on computation offloading - A survey", *Computer Communications*, vol. 159, pp. 133–154, 2020. `https://doi.org/10.1016/j.comcom.2020.05.001`

[7] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu, "Reducing energy per instruction via dynamic resource allocation and voltage and frequency adaptation in asymmetric multicores", in *2014 IEEE Computer Society Annual Symposium on VLSI*, 2014, pages 436–441, Doi: 10.1109/ISVLSI.2014.110.

[8] M. Zhu and K. Shen, "Energy discounted computing on multicore smartphones", in *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016, pages 129–141.

[9] J. Choi, B. Jung, Y. Choi, and S. Son, "An adaptive and integrated low-power framework for multicore mobile computing", *Mobile Information Systems*, vol. 2017, 2017. `https://doi.org/10.1155/2017/9642958`

[10] W. Sheng, J. Castrillon, and R. Leupers, "Software compilation and optimization techniques for heterogeneous multi-core platforms", *Multi-Processor System-on-Chip 2: Applications*, John Wiley & Sons, 2021, pages 203-231. ISBN: 978-1-789-45022-4.

[11] C. Kumar, K. Naik, et al., "Smartphone processor architecture, operations, and functions: current state-of-the-art and future outlook: energy performance trade-off: Energy–performance trade-off for smartphone processors", *Journal of Supercomputing*, vol. 77, no. 2, pp. 1377–1454, 2021. `https://doi.org/10.1007/s11227-020-03312-z`

[12] P. Pawar, S. Dhotre, and S. Patil, “CFS for addressing CPU resources in multi-core processors with AA tree”, *International Journal of Computer Science and Information Technologies*, vol. 5, no. 1, pp. 913-917, 2014.

[13] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, ”Parallel frame rendering: Trading responsiveness for energy on a mobile GPU”, in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, IEEE Press, 2013, pages 83–92. Doi: 10.1109/PACT.2013.6618806.

Received on January 11, 2022
Accepted on October 08, 2022