A User-Centric CPU-GPU Governing Framework for 3D Games on Mobile Devices

Wei-Ming Chen[1], Sheng-Wei Cheng[1], Pi-Cheng Hsiu[2,3], and Tei-Wei Kuo[1,2,3,4]

[1] Department of Computer Science and Information Engineering, National Taiwan University, Taiwan

[2] Research Center for Information Technology Innovation, Academia Sinica, Taiwan

[3] Institute of Information Science, Academia Sinica, Taiwan

[4] Graduate Institute of Networking and Multimedia, National Taiwan University, Taiwan

r02922032@csie.ntu.edu.tw, d02922004@csie.ntu.edu.tw, pchsiu@citi.sinica.edu.tw, ktw@csie.ntu.edu.tw

***Abstract*—Graphics-intensive mobile games are becoming increasingly popular, but such applications place high demand on device CPUs and GPUs. The design of current mobile systems results in unnecessary energy waste due to lack of consideration of application phases and user attention (a "demand-level" gap) and because each processor administers power management autonomously (a "processor-level" gap). This paper proposes a user-centric CPU-GPU governing framework which aims to reduce energy consumption without significantly impacting the user experience. To bridge the gap at the demand level, we identify the user demand at runtime and accordingly determine appropriate governing policies for the respective processors. On the other hand, to bridge the gap at the processor level, the proposed framework interprets the frequency scaling intents of processors based on the observation of the CPU-GPU interaction and the processor status. We implemented our framework on a Samsung Galaxy S4, and conducted extensive experiments with real-world 3D gaming apps. Experimental results showed that, for an application being highly interactive and frequent phase changing, our proposed framework can reduce energy consumption by 45.1% compared with state-of-the-art policy without significantly impacting the user experience.**

I. Introduction

Mobile apps have grown rapidly both in number and complexity, driving increased demand for computing and battery capacity of mobile devices. Among mobile apps, graphics-intensive games have become much more popular, with a recent market survey showing that game applications are by far the most popular downloads [1]. The interactive nature of games results in highly variable workloads [9, 10]. Moreover, game developers seek to improve the visual aesthetic quality of their products through increasing display density [11], resulting in increased computing demand. In response to these developments, many of the latest mobile devices are equipped with DVFS-enabled processors so as to reduce power consumption without significantly affecting user experience. However, in current mobile systems, each processor administers power management autonomously, leading to inefficiency. Under certain circumstances, the battery power is wasted without improving the user experience. Thus, this paper seeks to rethink power management designs for use in graphics-intensive game applications.

Along with the introduction of DVFS-enabled CPUs [15], significant research has been devoted to both theory and practice of energy-efficient scheduling. In theory, given that the real-time constraints of applications are known a priori, Yao et al. [16] provide an off-line optimal scheduling algorithm. However, in practice, it is frequently difficult to determine the exact execution characteristics of an application in advance. In response to this, commodity operating systems introduce, apart from a workload scheduler, a governor that monitors CPU loading during a scheduling period and adjusts the processors frequency accordingly. When different performance/energy tradeoffs are considered, governor policies [12] have been proposed with different frequency-scaling criteria. Investigations into governing DVFS-enabled CPUs has led to successful energy reduction, driving the proliferation of new DVFS-enabled processors and devices. The emergence of DVFS-enabled GPUs in the latest mobile devices allows for the efficient execution of graphics-intensive game applications.

Previous research into optimizing the energy efficiency of game applications has considered the perspectives of user interaction and workload prediction. For game applications which do not involve much user interaction, processor workloads depend heavily on game frames which offer a rich variety of structural information [9]. Game frames can be parsed to derive gaming workload characteristics, such as the texture operations [14] or the number of brush and alias models [8]. In this way, frequency scaling strategies for either a CPU [8, 9] or a GPU [14] have been proposed. On the other hand, things become more complicated when intensive user interaction is considered. This is because buffering frames in game applications is hardly possible due to its dependence on the user input. In view of this, algorithms aiming at the prediction accuracy were proposed for CPU frequency scaling [5–7] or the power reduction for wireless network interface [2], which periodically adjust workload predictions based on feedback of prediction errors. However, these approaches only consider the governance of CPU or a GPU loading in isolation.

Graphics-intensive game applications place high demands on both CPU and GPU. Therefore, managing these processors in isolation may result in an information gap between their respective governors (a processor-level gap) and may thus result in inefficient energy use. Previous works have proposed joint CPU and GPU governance [3, 13]. By adopting a cost model [13] or

978-1-4673-8388-2/15/$31.00 ©2015 IEEE

a power-performance model [3], they aim to achieve the target FPS range while reactively minimizing power consumption. However, both approaches focus only on the rendering phase of a gaming application, neglecting applications with frequent phase changes. This may result in a governing strategy that adversely impacts the user experience, because the user may care about different performance metrics at different phases [4] (a demand-level gap).

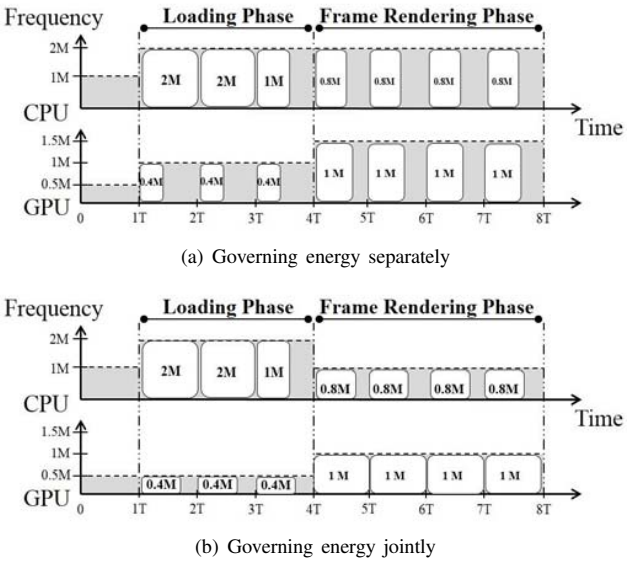
This paper proposes a governing framework which aims to reduce energy consumption without significantly impacting the user experience. To do so, the proposed governing framework should bridge both the demand-level and processor-level gaps. To bridge the demand-level gap, our framework considers both application phases and user attention. In turn, the governing policies for both the CPU and the GPU can be determined according to the classification of both application phases and user attention. On the other hand, to bridge the processor-level gap, it is important to communicate the frequency-scaling intents of the processors. Based on the observation of the CPU-GPU interaction and the processor status, the proposed framework can interpret the frequency-scaling intents of processors. In this way, we can prevent the governors from scaling frequency without improving the user experience. We implemented our framework on a Samsung Galaxy S4 which was used to conduct extensive experiments with real-world 3D gaming apps. Experimental results showed that, for an application being highly interactive and frequent phase changing, our framework can reduce energy consumption by 45.1% when compared with the state-of-the-art governor policy [13] without significantly impacting user experience.

The rest of this paper is organized as follows. Section II reviews the background of CPU/GPU interaction and points out the design challenges, while Section III details the design of the user-centric CPU-GPU governing framework. Section IV provides experimental results with a discussion of energy reduction and performance. Finally, Section V draws conclusions.

II. Background and Motivation

A. Background Information

With the introduction of GPUs, efficient graphics rendering is now accomplished through the cooperation of the CPU and the GPU in a system. For example, a typical 3D gaming workload contains at least two distinct phases: a loading phase and a frame rendering phase. In the loading phase, in which the CPU constructs the graphic data using complex game physics for 3D scenes and configuring the GPU for initialization (e.g., object loading, 3D scene configuring, etc). In the frame rendering phase, the GPU renders the frames according to the data delivered from the CPU in a best-effort fashion. In these two phases, an application will generate a series of *GPU commands* using libraries, compilers, and runtime frameworks. A GPU device driver contains a command buffer and a command dispatch unit. To transfer GPU commands from the CPU to the GPU, an application first pushes GPU commands to the command buffer. The GPU driver then notifies the GPU device



(a) Governing energy separately

(b) Governing energy jointly

Fig. 1. A motivational example

of the commands by configuring the command dispatch unit. In this way, workloads can successfully be dispatched to the GPU device. On the other hand, provided a DVFS-enabled device (e.g., CPU or GPU), a governor is used to monitor device utilization and scale the frequency to minimize energy consumption.

To adapt to different performance/energy tradeoffs based on user preferences, different policies [12] for scaling available processor frequencies have been proposed for a governor. This paper investigates the most common policies implemented in Linux. To optimize processor performance, the *performance* policy will set the processors frequency to the highest available level. To minimize energy consumption, the *power-saving* policy will set the processors frequency to the lowest available level. Furthermore, the *on-demand* and *conservative* policies are designed to adapt to different workloads on the fly. Both the on-demand policy and the conservative policy set lower and upper utilization thresholds. When the lower utilization threshold is reached, both the policies will scale down the processors frequency to the next available lower level. However, the two policies behave differently when the higher utilization threshold is reached. The on-demand policy will scale up the processors frequency to the highest available level, while the conservative policy will scale up the processors frequency to the next available higher level. Given these policies, a user can enforce a policy according to the users desired performance/energy tradeoff for governing the CPU or the GPU. However, since the workloads on both the CPU and the GPU can be revealed by GPU commands, a better policy can be obtained if we can govern the energy consumption of both the CPU and the GPU jointly.

B. Example of Benefits

Let us consider a scenario in which a 3D gaming application is running on a mobile device equipped with a CPU and

a GPU, as shown in Fig. 1. Assume that the considered CPU and GPU respectively support two and three distinct frequency levels. Moreover, the CPU operating at the highest and lowest frequency levels can respectively provide 2 mega and 1 mega computing cycles per scheduling period T. The GPU operating at the highest, middle, and lowest frequency levels can respectively provide 1.5 mega, 1 mega, and 0.5 mega computing cycles per scheduling period T. The gaming application contains exactly the two phases, a loading phase and a frame rendering phase. In the loading phase, the gaming application requires 5 mega CPU computing cycles in total, and 0.4 mega GPU computing cycles per period for GPU initialization. Whereas, in the frame rendering phase, the application requires 0.8 mega CPU computing cycles per period and 1 mega GPU computing cycles per period for fluent scene changes.

Figure 1(a) shows the result when both the CPU and the GPU adopt the conservative policy and govern their energy consumption separately. Suppose that the lower and the upper utilization thresholds of the conservative policy are 35% and 80%, respectively. Initially, both the CPU and GPU operate at their respective lowest available frequencies. When the 3D gaming application starts at time 1T, it first enters the loading phase. In the loading phase, the workload drives the CPU and GPU governors to scale up their respective frequency levels to provide 2 mega and 1 mega computing cycles per scheduling period. At time 4T, the loading phase is completed because the required total CPU computing cycles in this phase is achieved. Moreover, the frequency level of the CPU will remain unchanged in the following periods since the utilization of the workload on CPU exceeds the lower utilization threshold of the conservative policy. Meanwhile, the workload on GPU (1 mega cycles/period) will drive the GPU governor to scale up the frequency level of the GPU at 4T. In this way, both the CPU and GPU operate at their respective highest available frequencies most of the time under such a governing policy.

Now, suppose that we know the phases of the application a priori and are able to jointly configure the adopted governor policies for the CPU and GPU. Specifically, when the application is in the loading phase, the governing policies for the CPU and GPU are respectively configured to be the conservative and power-saving. Whereas, when the application is in the frame rendering phase, the governing policies for the CPU and GPU are respectively configured to be the power-saving and the conservative. Besides, the frequency of the GPU is upscaled to provide 1 mega computing cycles per scheduling period for the workload on GPU at 4T. The joint governing result is shown in Fig. 1(b). As can be seen, most of the time, the CPU and GPU operate at their respective lower available frequencies when compared with Fig. 1(a). From the comparison, we can see that the strategy that governs energy separately incurs many unnecessarily instances of frequency upscaling, which wastes energy. More specifically, the GPU operating at the lower frequency level can already provide enough computing cycles for the loading phase. Likewise, for the frame rendering phase, the CPU operating at the lower frequency level can already provide enough computing cycles.

The main challenges in joint CPU-GPU governing stem from two information gaps: *a demand-level gap* and *a processor-level gap*. The demand-level gap is due to the failure to consider user perception and application status. That is, user performance demands may vary in different application phases, such as the response time in the loading phase, or the frame rate in the frame rendering phase. On the other hand, the processor-level gap originates from the separated governors which, in turn, are adopted in legacy desktop systems and not redesigned for energy-constrained systems. *The objective in this paper is to reduce energy consumption while maintaining user experience by taking the performance metrics at different application phases into consideration.* Specifically, for the loading phase, we aim to reduce the response time while, for the frame rendering phase, we aim to achieve the application's default frame rate. In reality, the considered problem may become more severe given an increased number of different application phases, or number of different frame rates required at scene changing.

III. USER-CENTRIC CPU-GPU GOVERNING FRAMEWORK

A. Framework Overview

In this section, we propose a user-centric CPU-GPU governing framework to bridge both the demand-level and processor-level gaps. To bridge the gap at the demand level, it is essential to identify the user demand at runtime and accordingly determine the appropriate governing policies for respective processors. Correspondingly, to bridge the gap at the processor level, it is important to communicate the frequency-scaling intents of the processors. In this way, we can prevent governors from scaling frequency while not improving user experience. Note that, rather than reinventing the wheel, the proposed governing framework is structured above the governor policies, thus existing governor policies can be integrated into the proposed framework when needed.

As shown in Figure 2, our governing framework[1] is comprised of a *user demand classifier*, a *unified policy selector*, and a *frequency-scaling intent communicator*. The user demand classifier is activated periodically to monitor the game application at runtime, while the unified policy selector configures the policy of each processor governor. In turn, the frequency-scaling intent communicator is activated only when the utilization threshold for scaling frequency is reached at either the CPU or the GPU. In Section III-B, to classify the user demand, we differentiate the application phases based on GPU commands, and determine user attention according to the occurrence of any touch event. After classifying user demand, we then determine the governing policies in Section III-C and classify the frequency-scaling intents of the processors in Section III-D.

[1]For portability across different Android platforms, we implement most of the proposed design in the user space, and minimize the modifications of kernel codes by adding several hooks to the GPU driver and the CPU governor in the kernel space.

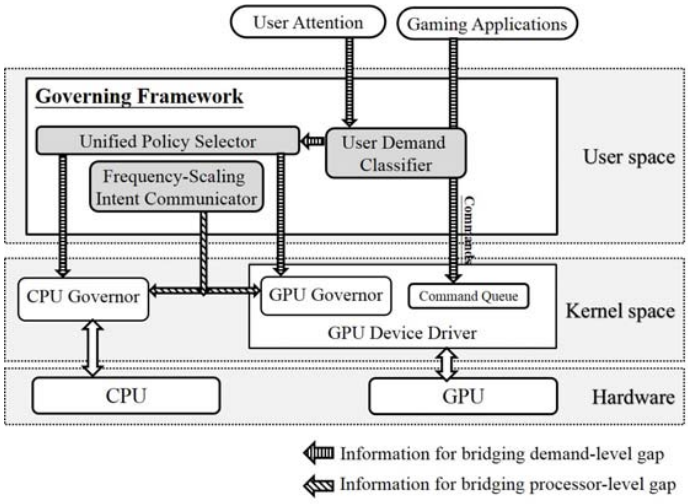


Fig. 2. The governing framework.

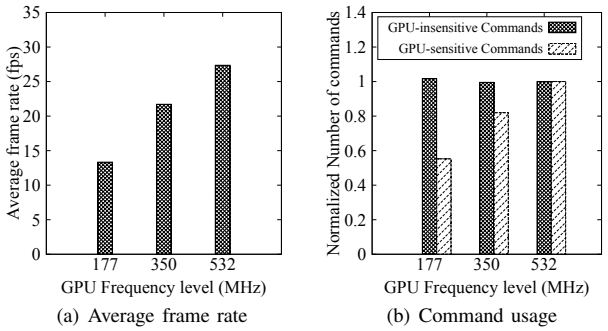


Fig. 3. Statistics of the benchmark "Ice Storm Unlimited"

B. User Demand Classification

To identify the user demand, we should take both application phases and user attention into consideration. To classify application phases, we first provide the observation derived from profiling the GPU commands, and accordingly differentiate between an application's *GPU-sensitive* phase (e.g., the rendering phase) and its *GPU-insensitive* phase (e.g., the loading phase). In turn, according to the extent to which a user interacts with a game application, we differentiate between the user's interacting and non-interacting states.

The interaction between a CPU and a GPU can be characterized by using GPU commands. Moreover, different commands may stress either CPU computing or GPU computing. Commands that put much more emphasis on the GPU computing are used for 3D frame rendering. Meanwhile, commands that put much more emphasis on the CPU computing are used for transferring graphical data (e.g., texture or 3D object model) to GPU memory, maintaining the communication protocol (e.g., waking up the GPU before sending commands), and so forth. According to observed commands with different computing resource emphases, we classify the commands as being GPU-sensitive and GPU-insensitive.

For a typical gaming application, application phases that are susceptible to GPU frequency scaling can be identified by monitoring the GPU commands at runtime. We observed the average frame rates with varying GPU frequencies and the number of invoked commands while executing a gaming application[2]. The results are shown in Figure 3. As can be seen from Figure 3(a), the average frame rates increased with GPU frequency. However, things are different for the number of invoked commands in Figure 3(b). To compare the results of different types of commands, the number of invoked commands for GPU-sensitive or GPU-insensitive is normalized with respect to the number of GPU-sensitive or GPU-insensitive invoked commands with a GPU frequency of 532 MHz. According to the figure, the number of invoked GPU-sensitive commands was susceptible to the change in GPU frequency, while the number of invoked GPU-insensitive commands was not.

Based on our observations, we classified the application phases using different command types. Moreover, at runtime, the invoked commands should be investigated periodically so as to quickly determine the application phases. In this way, the phase is considered GPU-insensitive if, in the last period, we observed commands emphasizing CPU computing or no command at all. Otherwise, the phase is considered GPU-sensitive. Following this classification, we can easily discriminate the phases of a 3D gaming application. Meanwhile, for applications executed only on the CPU, such a phase classification would degenerate into a GPU-insensitive phase only.

To properly identify user demand, user attention should be considered as well. This is because, when a user is interacting with an application, the performance requirement is elevated to respond to input events. Therefore, the framework classifies user attention into interacting and non-interacting states by determining whether any touch event occurs[3]. Till now, the framework can discriminate the state of user attention and the application phase. Accordingly, the processor management should conform the identified user demands so as to save energy without degrading user experience.

C. Unified Policy Selection

User Attention	Application Phase	
	GPU-insensitive	GPU-sensitive
Non-Interacting	C: On-demand G: Conservative	C: Conservative G: Conservative
Interacting	C: Performance G: On-demand	C: On-demand G: Conservative

TABLE I. POLICY SELECTION CLASSIFICATION

Once the changes in the user demand have been determined, we can then proceed to determine the governing policy for each

[2] 3DMark, http://www.3dmark.com/

[3] For example, in Linux, to catch the user touch events, we can utilize the touchscreen driver in the Proc file system.

processor. By combining the phases of the game application and the states of the user, we end up with four distinct classes for classifying the user demand. Note that, in this paper, we focus our attention on how the framework should react according to the change from the user demand, rather than how to gracefully classify the user demand. The latter is in itself an interesting research issue but lies outside the scope of this paper.

As mentioned in Section II, different application phases imply different performance metrics of concern to the user. For example, when a gaming application is in the GPU-insensitive phase, the application is often conducting graphics object loading. Therefore, at this phase, the application workload should be finished as soon as possible, otherwise, the user experience will be adversely affected by the delayed response. To this end, the governing policy on the CPU should be configured to favor performance, while the governing policy on the GPU should emphasize energy efficiency. On the other hand, when a gaming application is in the GPU-sensitive phase, the application is often conducting graphics object rendering, which is conducted with consideration of user perception. That is, the content should be displayed on the screen for a long enough duration to be perceived by the user, so a moderate number of scene changes occur within a scheduling period. Thus, at this stage, the governing policy on both the CPU and GPU should be configured to emphasize energy-efficiency.

Table I summarizes our unified governing policy. Specifically, when the application is in the GPU-insensitive phase and the user is not interacting with the application, we should configure the CPU governor to the on-demand policy and the GPU governor to the conservative policy. On the other hand, when the application is in the GPU-insensitive phase and the user is interacting with the application, we should configure the CPU governor to the performance policy and the GPU governor to the on-demand policy. In this way, the events caused by user interaction will be processed as soon as possible in the CPU so as to provide the best possible user experience. In addition, when the application phase switches to GPU-sensitive following user interaction, the governing policy on the GPU should be configured to emphasize increased performance. Likewise, when the application is in the GPU-sensitive phase and the user is not interacting with the application, we should configure both the CPU and GPU governors to the conservative policy. On the other hand, when the application is in the GPU-sensitive phase and the user is interacting with the application, we should configure the CPU governor to the on-demand policy and the GPU governor to the conservative policy to ensure that the CPU can react quickly to user interaction.

D. Frequency-Scaling Intent Communication

Based on the observation of the CPU-GPU interaction and the processor status, the proposed framework can interpret the frequency-scaling intents of the processors. In this way, the frequency-scaling decision of a processor's governor will be abolished if the decision does not conform to the intent of the other processor's governor. That is, when the utilization threshold of the policy adopted in the GPU (resp. CPU) is reached, we are now required to know the frequency-scaling intent from the CPU (resp. GPU) to finalize the scaling decision at the GPU (resp. CPU). We classify the intent as *"speedup"*, *"slowdown"* and *"self-rule"*. When a governor receives a "speedup" intent, it needs to scale up its processor's frequency. Whereas a governor receives a "slowdown" intent, it is not allowed to scale up its processor's frequency. Otherwise, when the "self-rule" intent is received, a governor can administer its processor autonomously. In summary, such a design prevents governors from scaling frequency without improving the user experience.

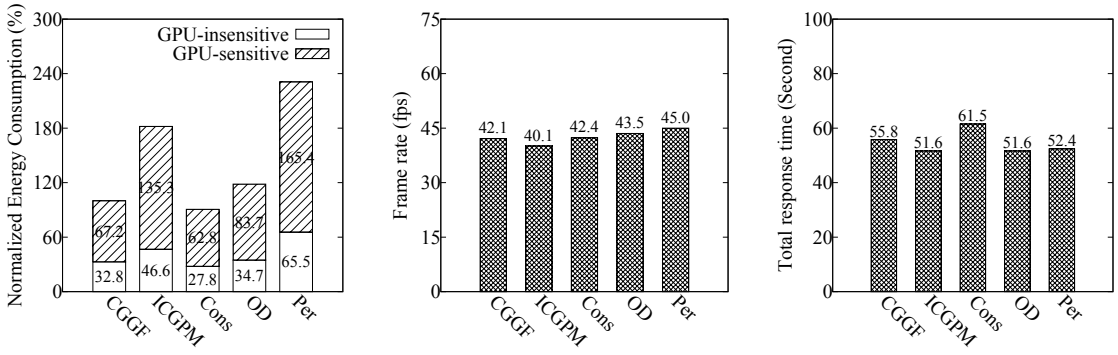
Below, we provide a few examples of configuring the frequency-scaling intents when a processor is overloaded, a governor is about to scale down the frequency level, or the user perception has been satisfied. When the CPU is already overloaded, the workload on the GPU will not increase since the workload is assigned from the CPU. In this way, the intent from the CPU governor is "slowdown". In contrast, when the GPU is overloaded during a GPU-sensitive phase, the GPU will send a "slowdown" intent to the CPU governor since boosting the CPU will not increase the frame rate. In addition, governors will actively send "slowdown" intents when they are going to scale down the frequency level of the corresponding processors. This is because, if the governor dominating the user experience scales down the frequency level of the processor, increasing the frequency of the other processor does not help improve the user experience. For example, in the GPU-insensitive phase, when the CPU scales down its frequency level, the CPU governor sends a "slowdown" intent to the GPU governor. Moreover, user perception can also be considered by using intents. That is, when the frame rate of a gaming application reaches the hardware refresh rate set by the operating system[4], providing more computing resources will not help improve the user experience. Therefore, the framework should send "slowdown" intents to the governors on behalf of the user.

IV. Performance Evaluation

A. Experimental Setup

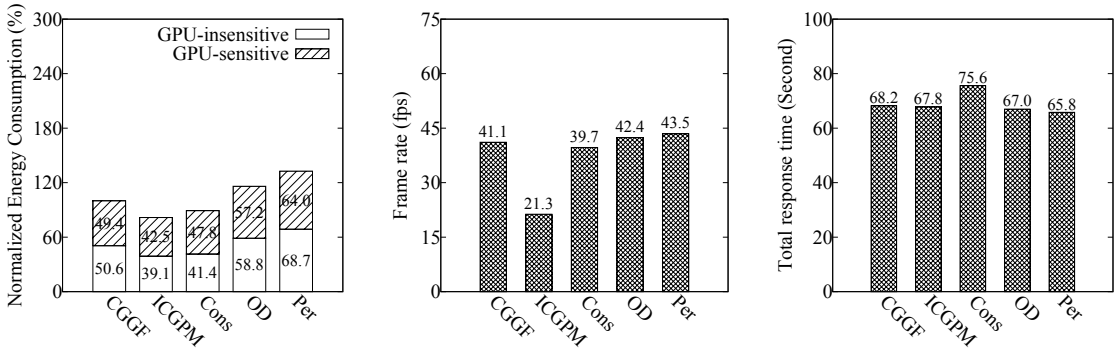
To evaluate the proposed framework, we conducted extensive experiments on a commercial Android smartphone with real-world 3D gaming apps. Specifically, the experiments were conducted on a Samsung Galaxy S4 equipped with a CPU allowed to operate at 17 frequency levels and a GPU at 6 frequency levels. When a network connection is needed, the smartphone can access the Internet via a dedicated 802.11n access point (TP-LINK TL-WR841N). To practically profile the CPU-GPU interaction, the period to investigate the invoked GPU commands was set to 1 second. This is because scene changing in games usually needs to proceed long enough to be perceived by users. As for measuring the smartphone's energy consumption, we used the power monitor produced by Monsoon Solutions. In turn, the workload energy consumption was calculated by subtracting the smartphone's energy consumption in idle state. In addition, to obtain further more insight into the

[4]Vsync, https://source.android.com/devices/graphics/implement.html



(a) Energy consumption (b) Average frame rate (c) Total response time

Fig. 4. High-interactive and frequent-switch scenario



(a) Energy consumption (b) Average frame rate (c) Total response time

Fig. 5. Low-interactive and frequent-switch scenario

trade-off, we also investigated the performance of the evaluated games. To this end, the performance metric for applications in the GPU-sensitive phase is the average frame rate while the performance metric for applications in the GPU-insensitive phase is the total response time. As astute readers might point out, phases in a game can change intermittently. Taking this fact into consideration, for the game under evaluation, the frame rate was averaged throughout all the GPU-sensitive phases while the total response time was the sum of all the GPU-insensitive phases.

	Interacting ratio(%)	Avg. Switching period (sec)	Total exec time (sec)
Trail Extreme 3	73	4.3	200
Sniper 3D Assassin	26	4	129
Real Football 2013	80	14.5	248
Worldcraft	20.8	23	115

TABLE II. THE DETAILS OF THE SELECTED GAMING WORKLOADS

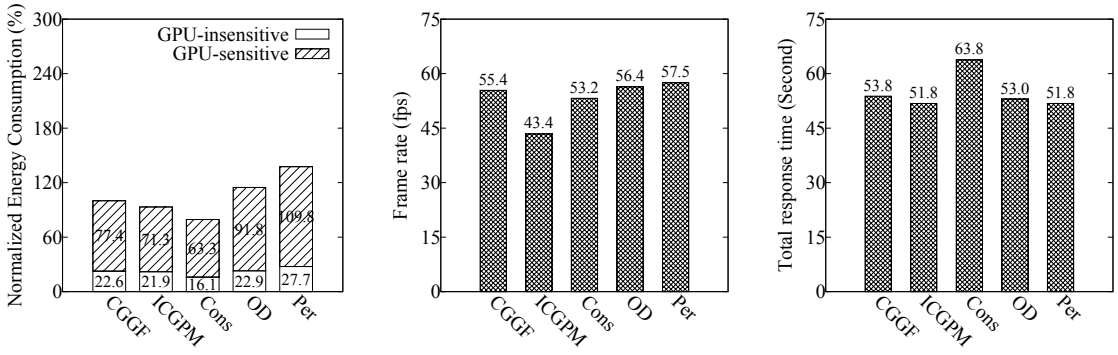
For gaming application workloads, we studied all possible combinations of four popular 3D game apps. Table II shows the profiling results of the applications, and we classified their types according to the extent to which a user interacts with the game and how often the phases of the game switch. A user playing Trail Extreme 3[5] needs to balance the virtual motorcycle by touching the device screen in a timely fashion and aim to reach different destinations (or scenes) shortly. Provided these characteristics, Trail Extreme 3 was classified as a *high-interactive* and *frequent-switch* workload. A *low-interactive* and *frequent-switch* workload was characteristic of Sniper 3D Assassin[6] due to frequent scene changes. Note that the reason we classify Sniper 3D Assassin as a low-interactive workload is that it does not incur extra computing when a user tilts the device. Real Football 2013[7] was characterized as *high-interactive* and *infrequent-switch* because the user needs to frequently interact with the device to control the football players in a virtual football field. Worldcraft was characterized as *low-interactive* and *infrequent-switch* because allowing the user to casually explore and construct the preloaded 3D virtual world entails a long phase-switching time.

To show the effectiveness of our proposed approach, policies including the conservative, the on-demand, the performance and ICGPM [13] are applied for comparison. For those tunable parameters of the first 3 policies, we followed the designs of the CPU governor in the Samsung Galaxy S4, and simultaneously applied them in both the CPU and GPU governors. On the other hand, for the ICGPM policy, it detects phase changing

[5]http://www.deemedya.com/

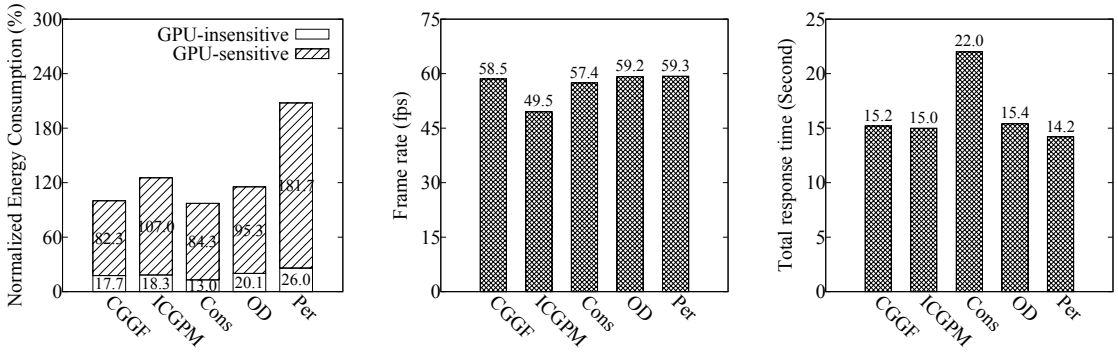
[6]http://www.fungames-forfree.com/

[7]http://www.gameloft.com/



(a) Energy consumption (b) Average frame rate (c) Total response time

Fig. 6. High-interactive and infrequent-switch scenario



(a) Energy consumption (b) Average frame rate (c) Total response time

Fig. 7. Low-interactive and infrequent-switch scenario

in a game by using both the CPU and GPU utilizations. In GPU-insensitive phases, the ICGPM policy maximizes the CPU frequency and minimizes the GPU frequency. Moreover, in GPU-sensitive phases, the ICGPM policy jointly governed the CPU and GPU so as to assure the minimum FPS requirement. In addition, to fairly compare different policies, we recorded user interaction in each game by logging their touch events[8]. For each gaming workload, the experiment results are the average of 5 independent experiment measurements of the same input trace. In addition, for ease of readability, the policies of the proposed framework, the conservative, the on-demand and the performance are respectively referred to as the CGGF, Cons, OD, and Per.

B. Experiment Results

For each type of the workloads, we investigate the energy consumption and the performance in different phases by applying different governing policies. The energy consumed in the GPU-sensitive and GPU-insensitive phases was measured separately[9] while the final results were all normalized with respect to the total energy consumed by using the CGGF policy. On the other hand, the average frame rates in the GPU-sensitive phases and the total response time in the GPU-insensitive phases were also investigated.

For workloads that switch phases frequently, Figures 4 and 5 respectively show the experimental results with respect to high-interactive and low-interactive workloads. As can be seen in the figures, the Per and OD policies performed slightly better than our framework in terms of average frame rate (resp. +2.9 fps and +1.4 fps) and total response time (resp. -3.4 sec. and -4.2 sec.) for the high-interactive workload. For the low-interactive workload, likewise, the Per and OD policies performed slightly better than our framework in terms of average frame rate (resp. +2.4 fps and +1.3 fps) and total response time (resp. -2.4 sec. and -1.2 sec.). However, our framework significantly outperformed Per and OD policies in terms of energy consumption, with Per and OD policies consuming +131% and +18.4% more energy, respectively, for the high-interactive workload and +32.7% and +16%, respectively, for the low-interactive workload. Moreover, as expected, the Cons policy consumed less energy than our framework for both the high-interactive workload (-9.7%) and the low-interactive workload (-10.7%). In response to such energy saving, the performance of the Cons policy was worse than our framework in terms of the total response time (+5.7 sec. for the high-interactive workload and +7.4 sec. for the low-interactive workload).

[8]HiroMacro Auto-Touch Macro, http://prohiro.com/

[9]We logged the GPU commands at runtime and, accordingly, attributed the measured energy consumption to different phases.

Interestingly, in terms of energy consumption, our framework outperformed the ICGPM policy in the high-interactive workload, but underperformed in the low-interactive workload. To see the whole picture, we should consider the average frame rate and the total response time. For the high-interactive workload, the average frame rate of the ICGPM policy was slightly degraded (-2 fps) and the total response time was slightly improved (-4.2 sec.), whereas the incurred energy consumption was significantly increased (+81.9%). On the other hand, for the low-interactive workload, the incurred energy consumption was reduced (-18.4%), whereas the average frame rate of the ICGPM policy was significantly degraded (-19.8 fps) and the total response time was about the same. In other words, when using the ICGPM policy, the user experienced a perceptible delay during the experiments. This is because the ICGPM policy calculated a current frame rate requirement based on historic frame rates, and minimized energy consumption according to the calculated frame rate requirement.

Figures 6 and 7 respectively show the experimental results with respect to high-interactive and low-interactive workloads that switch phases infrequently. Similarly, for both workloads, the Per and OD policies performed better than our framework, but our framework provided significant advantage in terms of energy consumption. In addition, the Cons policy consumed less energy than our framework for both the high-interactive workload (-20.6%) and the low-interactive workload (-2.7%). However, the performance of the Cons policy was worse than our framework in terms of the total response time (+10 sec. for the high-interactive workload and +6.8 sec. for the low-interactive workload). As for the ICGPM policy, in terms of energy consumption, our framework outperformed the ICGPM policy in the low-interactive workload, but underperformed in the high-interactive workload. That is, the additional energy consumption was +24.9% for the low-interactive workload and -6.9% for the high-interactive workload. However, for both the low-interactive workload and the high-interactive workload, the average frame rates were all degraded (resp. -9 fps and -12 fps).

V. Concluding Remarks

In this paper, we propose a user-centric CPU-GPU governing framework to reduce energy consumption without significantly impacting user experience when using graphics-intensive mobile games. To this end, the proposed governing framework bridges both the demand-level and processor-level gaps. To bridge the demand-level gap, our framework takes both application phases and user attention into consideration. The governing policies for both the CPU and the GPU can be determined according to the classification of both application phases and user attention. On the other hand, to bridge the processor-level gap, the proposed framework can interpret the frequency-scaling intents of processors based on observation of the CPU-GPU interaction and the processor status. We implemented our framework on a Samsung Galaxy S4, and conducted extensive experiments with real-world 3D gaming apps. Experimental results showed that, for an application being highly interactive and frequent phase changing, our framework can reduce energy consumption by 45.1% when compared with the state-of-the-art policy without significantly impacting the user experience.

Acknowledgement

This work was supported in part by the Ministry of Science and Technology under Grants No. 103-2622-E-002-034- and No. 102-2221-E-001-007-MY2.

References

[1] Most popular Apple App Store categories in March 2015. http://www.statista.com/statistics/270291/popular-categories-in-the-app-store/.

[2] B. Anand, A. L. Ananda, M. C. Chan, L. T. Le, and R. K. Balan. Game action based power management for multiplayer online game. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, MobiHeld '09, pages 55–60, New York, NY, USA, 2009. ACM.

[3] A. P. T. M. Anuj Pathania, Alexandru Eugen Irimiea. Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs. In *Proc. of IEEE/ACM DAC*, 2015.

[4] B. Dietrich and S. Chakraborty. Power management using game state detection on android smartphones. In *Proc. of ACM MobiSys*, 2013.

[5] B. Dietrich and S. Chakraborty. Forget the battery, let's play games! In *Proc. of IEEE ESTIMedia*, 2014.

[6] B. Dietrich, S. Nunna, D. Goswami, S. Chakraborty, and M. Gries. LMS-based Low-Complexity Game Workload Prediction for DVFS. In *Proc. of IEEE/ACM ICCD*, pages 417–424, 2010.

[7] Y. Gu and S. Chakraborty. Control Theory-based DVS for Interactive 3D Games. In *Proc. of IEEE/ACM DAC*, pages 740–745, 2008.

[8] Y. Gu and S. Chakraborty. Power Management of Interactive 3D Games Using Frame Structures. In *Proc. of IEEE VLSID*, pages 679–684, 2008.

[9] Y. Gu, S. Chakraborty, and W. T. Ooi. Games Are Up for DVFS. In *Proc. of IEEE/ACM DAC*, pages 598–603, 2006.

[10] B. Mochocki, K. Lahiri, and S. Cadambi. Power Analysis of Mobile 3D Graphics. In *Proc. of IEEE DATE*, pages 502–507, 2006.

[11] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen. Mobile GPU Power Consumption Reduction via Dynamic Resolution and Frame Rate Scaling. In *Proc. of USENIX HotPower*, pages 5–5, 2014.

[12] V. Pallipadi and A. Starikovskiy. The Ondemand Governor: Past, Present, and Future. In *Proc. of Linux Symposium*, pages 223–238, 2006.

[13] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU Power Management for 3D Mobile Games. In *Proc. of IEEE/ACM DAC*, pages 40:1–40:6, 2014.

[14] B. Sun, X. Li, J. Song, Z. Cheng, Y. Xu, and X. Zhou. Texture-Directed Mobile GPU Power Management for Closed-Source Games. In *Proc. of IEEE HPCC*, pages 348–354, 2014.

[15] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proc. of USENIX OSDI*, 1994.

[16] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. of IEEE FOCS*, pages 374–382, 1995.