# Usage History-Directed Power Management for Smartphones

Xianfeng Li(✉), Wen Wen, and Xigui Wang

School of Electronic and Computer Engineering, Peking University, Shenzhen 518055, China
lixianfeng@pkusz.edu.cn, {wenwen,wangxigui}@sz.pku.edu.cn

**Abstract.** Smartphones are now equipped with powerful application processors to meet the requirement of performance demanding apps. This often leads to over-provisioned hardware, which drains the battery quickly. To save energy for battery-powered smartphones, it is necessary to let the processor run at a power-saving state without sacrificing user experience. Android has implemented a set of CPU governors by leveraging dynamic voltage and frequency scaling (DVFS) according to the computational requirements of apps. However, they commonly adopt very conservative policies due to limited information, therefore leaving considerable energy reduction opportunities unexplored; or they are not responsive to user interactions, leading to poor user experiences. In this work, we start from an important observation on the repetitive patterns of smartphone usage for each individual user, and propose UH-DVFS, a usage history-directed DVFS framework leveraging on this observation. UH-DVFS can identify repetitive user transactions, and store their execution history information within a table. When such a user transaction is launched again, the table is consulted for an appropriate CPU frequency adaptation to reduce the energy consumption of the user transaction without sacrificing user experience. We have implemented the proposed framework on Android smartphones, and have tested it with real-world interaction-intensive apps. The results show that our framework can save energy from 10 % to 36 % without affecting the quality of user experiences.

**Keywords:** Smartphone · Android · Energy reduction · DVFS · User transaction

## 1 Introduction

The Application Processors (AP) used in today's smartphones are chasing the CPUs in PC domain to meet the increasing performance need of smartphone apps. For example, ARM Cortex A15 has adopted almost all architectural features (superscalar, out-of-order execution, branch prediction, multithreading, etc.) that were only found in traditional high-performance processors. However, unlike desktop computing, smartphones are battery-driven, and these


G. Wang et al. (Eds.): ICA3PP 2015, Part I, LNCS 9528, pp. 288–302, 2015.
DOI: 10.1007/978-3-319-27119-4_20

power-hungry features will drain the battery very quickly. To address this problem, some power-saving techniques, such as dynamic voltage and frequency scaling (DVFS), are commonly employed in mobile processors. The DVFS technique where the clock frequency of a processor is decreased to allow a correspongding reduction in the supply voltage [9]. According to the formula (1), where P is the dynamic power dissipated by the chip, C is the capacitance of the transistor gates, f is the operating frequency and V is the supply voltage. Here $\alpha$ shows the switching activity. a processor operating at a lower frequency will consume significantly less power.

$$P = \alpha C V^2 f \tag{1}$$

A difficult problem with DVFS-enabled processors is when and how to adapt the processor's frequency, such that the energy consumption is reduced without sacrificing the performance requirement of an application or system. In the context of smartphones, as the typical usage pattern is bursty user interactions, predicting the performance requirement becomes even more difficult. Therefore, exploiting the DVFS mechanism on a smartphone appropriately is a non-trivial issue. In current Android systems, a set of power management policies are enforced to make use of DVFS. However, for both the default policy named *Ondemand Governor* [10] and the *Interactive Governor*, a naive algorithm is used to adapt the processor's frequency: anytime the CPU utilization exceeds a specified threshold UP_THRESHHOLD, the CPU frequency is scaled up to the highest level. This conservative mechanism is clearly biased towards performance instead of energy, it might be the case that the computational capability is over-provisioned, and a less aggressive frequency setting might be enough for the computational requirement.

As the interactive usage of the smartphone contributes about half of its energy consumption [7], and current techniques such as the Android DVFS governors are not exploiting the energy-saving opportunities sufficiently, there is a strong need to investigate more aggressive and accurate techniques. In this work, we focus on the optimization of interactive energy consumption for smartphones, and propose a novel DVFS-based power management framework exploiting the repetitive patterns of the usage history.

Our work starts with a key concept called *user perceived transaction* (UPT), which is firstly defined in [16]. A UPT begins with a user input on the device (e.g., a screen touching or scrolling) and ends with the last display update when all the visible contents affected by this user input are drawn. Typically, an input from the end user may trigger a series of display updates, and the response time of an interactive session has a great impact on the quality of the user experience. Despite the importance of UPT latencies, there is little research that studies the tradeoff between user transaction performance and its energy consumption. By measuring the energy consumption of a smartphone at different usage scenarios, we gain two observations. First, different usage scenarios may have very different computational requirements. Second, for the same usage scenario that repeatedly happens, the UPTs exhibit very similar CPU usage patterns.

Based on these observations, we propose a usage history-directed DVFS scaling framework called UH-DVFS. The basic idea is to first capture the frequently invoked usage scenarios, and measure its computational requirement in the course of its execution, which will be recorded in a cache-like data structure, called User Transaction Table (UTT). Each time a user perceived transaction is triggered and detected, the UTT is queried for its computational requirement in its previous invocations. Upon a match, the recorded computational requirement will be used for making appropriate DVFS scaling, such that the performance can be met with the lowest possible energy consumption. We have implemented this framework on Android-powered smartphones, and have validated it with real-world interaction-intensive apps. The results show that our framework can save energy from 10 % to 36 % without affecting the quality of the user experience.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 describes the proposed framework in details. Section 4 evaluates the effectiveness of the framework with experimentation. Finally, Sect. 5 draws conclusions on this paper.

## 2 Related Work

Battery life of smartphones has been an important research topic in recent years. The first step for power and energy optimization is to understand how energy is consumed on smartphones. For this purpose, some research groups have studied usage patterns of smartphones by users to guide power optimization [6,14]. Another thread is to study the power/energy consumption of apps and smartphone components with power modeling [12,17] and power estimation [11] techniques. They have not only provided us with useful tools, but also have obtained some very important insights on power/energy consumption of smartphones.

For power/energy optimization, different techniques have been proposed for various hardware components of the smartphone. As our work in this paper is mainly concerned with CPU energy reduction, we only survey the related work on this component, which is one of the primary sources of energy consumption. Dynamic voltage and frequency scaling (DVFS) has been a common technique used for processor energy reduction. For example, the power management component in Linux implements some DVFS schemes like the *ondemand governor* [10], which increases the clock frequency when CPU load is above some threshold.

There are two metrics for evaluating the quality of a DVFS scaling technique: the *responsiveness* and *appropriateness*. Responsiveness means responding to the change of performance requirement in real-time, and appropriateness means that the scaled frequency is neither over-provisioned nor inadequate for the performance requirement. The techniques proposed in [8,15] are able to adapt to the user activity quickly. They scale to the max frequency indiscriminately when the user is interacting with the phone, and lower the frequency back to the idle level while users are thinking. The problem is that they are not able to predict the frequency appropriate for the user interaction, and conservatively scale

to the max frequency, which often wastes considerable energy. Another approach tries to predict the appropriate frequency with a resource-driven DVFS [5], which explores the correlation between CPU utilization and other resources. But this resource-driven scheme is unable to raise the clock frequency immediately in response to a user interaction, as it needs a time window to measure and compute the actual performance requirement. This is quite unfavorable for smartphone interactions, which are often bursty (lasting only a short amount of time). Therefore it does not satisify the *responsiveness* metric.

Unlike these techniques, our approach meets *responsiveness* and *appropriateness* simultaneously, it can quickly scale the frequency to an appropriate level for repetitive user interactions by exploiting history information.

## 3 Usage History-Directed DVFS Scaling

In this paper, we propose a usage history-directed DVFS scaling framework (UH-DVFS), which run repetitive user perceived transactions with a capped frequency best matching its computational requirement estimated from past execution histories.

### 3.1 Key Concepts and Principles

We first lay the foundation of our work by introducing a set of important concepts, notations, and key principles backing our framework.

**Activity.** It is one of the primary components offered by Android application framework; an activity provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map.

**User Perceived Transaction (UPT).** It is first introduced in [16], as shown in Fig. 1. A UPT begins with the user's manipulation of the device (e.g., a screen touching or scrolling) and ends with the last display update when all the visible contents affected by this user input are drawn. Typically, an input from the end user may trigger a series of display update, and the response time of an interactive session has a great impact on the quality of user experience. Note we also use the term **user transaction** for short in some places in this paper.

**Usage Scenario.** It characterizes a UPT with the activity and the corresponding user action on this activity. It can be viewed as an abstraction of a UPT, without details on the procedure of the UPT. A usage scenario can be formally represented as a tuple of $< activity, action >$.

**CPU Load.** It is defined as the percentage of non-idle CPU time in a duration. Modern operating systems all provide mechanisms and tools for reporting the CPU utilization in real-time. Usually a CPU load above 80 % indicates that the CPU is running very heavy tasks. CPU load is also called **CPU utilization** in the literature.

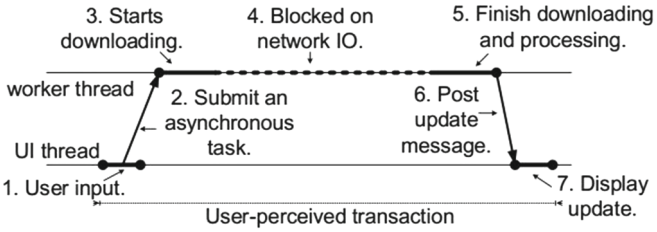



**Fig. 1.** An example of user perceived transaction ([16])

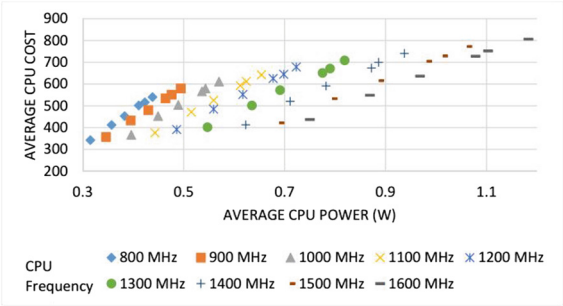



**Fig. 2.** Correlation between CPU cost and average CPU power ([13])

**Application Cost.** It is defined as the multiplication of the CPU load and the CPU frequency [4]. Intuitively, it quantifies the computational requirement of an application (or a segment of it) without the assumption of a specific frequency. Note this is only a rough estimation, as frequency is not the only aspect that decides the performance of a CPU, other factors, like the memory behavior, also has an impact. It is also called **CPU cost** in some literature [13].

**Normalized CPU Load.** Although CPU load is a good metric for characterizing how busy the CPU is, it does not fully exhibit the potential of the CPU. For example, a user may think that 100 % CPU load means the CPU is unable to meet the performance requirement of the application, but if this CPU load is reached with the CPU operating under a low frequency, we may easily satisfy the application by increasing the CPU frequency. Therefore, we use a more meaningful metric, called normalized CPU load, in many situations in this work. It is defined as follow (2) equation.

$$U_{norm} = U * (f_c / f_{max}) \tag{2}$$

where $U$ is the original CPU load, $f_c$ is the operating CPU frequency under which $U$ is measured, $f_{max}$ is the highest frequency offered by the CPU, and $U_{norm}$ is the normalized CPU load. For example, if the max CPU frequency $f_{max}$ is 1000MHz, but it runs at 300 MHz and get a 40 % CPU load, then the

normalized CPU load will be calculated as $40\,\% \times \frac{300MHz}{1000MHz} = 0.12$ normalized CPU load.

Our work is based on the following principles.

**(1) Invariance of CPU Cost.** The CPU utilization roughly scales linearly with the CPU core frequency for executing the same amount of work (e.g., a UPT), therefore, the CPU cost is maintained as a constant irrespective of the CPU frequency. For example, when the CPU frequency is increased by 50 %, its CPU load will decrease for about 50 % compared to its original number, and the CPU cost is the same for the two settings.

**(2) Correlation Between CPU Cost and Power Cost.** The authors in [13] conducted a comprehensive test, and the results show that with the CPU cost unchanged, lowering the CPU frequency can result in more power efficient execution without sacrificing user experiences, as shown in Fig. 2.

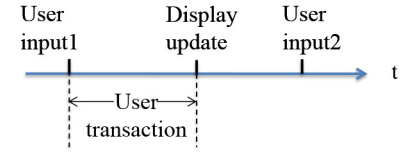
### 3.2 Observations on CPU and Power Behaviors of UPTs

We first conduct experiments by observing the CPU and power behaviors of user perceived transactions. We connect a Xiaomi-2S smartphone's battery interface to a Monsoon power monitor to collect the power trace of the smartphone, and use a tool offered by Qualcomm to capture the CPU behavior. Figure 3 is the result on a sequence of continuous user transactions.

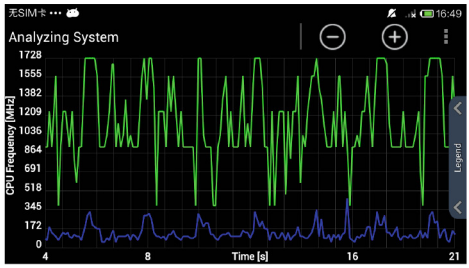
We first observe that the CPU frequency is adapted in a sub-second granularity, as shown in Fig. 3(b). The green line on the upper side is the trace for the actual frequency, and the blue line on the lower side shows the corresponding CPU load. As expected, the Ondemand governor of Android raises the frequency swiftly whenever it detects a high CPU load, such that sufficient CPU computation capability is provided for a heavy application load. More closely, we observe that most often, the CPU frequency simply jumps to the highest level $f_{max}$ by the Ondemand governor of Android system. But in reality, not all user transactions need such a high frequency, therefore wasting unnecessary energy.

But on the other hand, knowing the actual computational requirements of a user transaction beforehand is a non-trivial problem. First, a user transaction is just a short burst of execution, any reactive techniques that adjust the CPU frequency according to real-time guaging of the CPU load is unrealistic. On the other hand, predictive techniques that use the information of the past execution time window will not work either, because the CPU loads of different user transactions are different, and it would be meaningless to make decisions based on the recent execution.

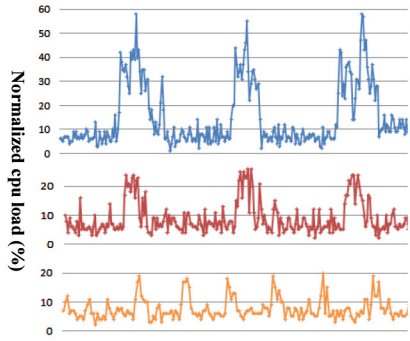
In stead of using reactive or native predictive methods, we propose a novel predictive method, which is based on the execution histories of the same usage scenario. This approach is based on an important observation: **the computational need for the same usage scenario is relatively stable for most usage scenarios**. As shown in Fig. 3(c), there are three usage scenarios (in color blue, red, and orange respectively). For each scenario, we repeat its user transaction for a couple of times. We can make two observations from them: first, each individual usage scenario has different CPU load; second, within the



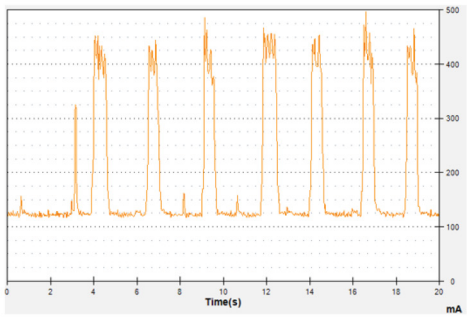

(a) user perceived transactions




(b) Variation of CPU frequency and CPU load




(c) Variation of normalized CPU load




(d) Variation of power consumption

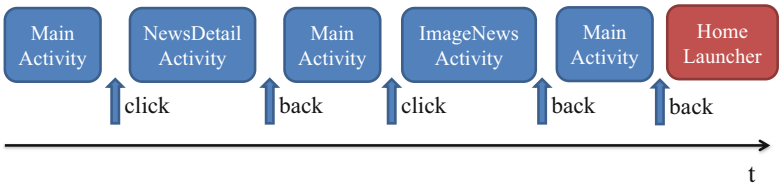
**Fig. 3.** CPU and power behaviors of user perceived transactions

same usage scenario, the CPU loads of different user transactions are quite similar. Therefore, we can use the history of the same usage scenario to predict its computational requirement of its current instance.

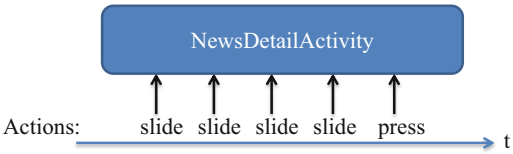
### 3.3 The UH-DVFS Scaling Framework

To use the execution history of a usage scenario, the first step is to record it. As defined in Sect. 3.1, a usage scenario is characterized by a tuple $< activity, action >$, where an *action* triggered by the user is enforced on the foreground *activity*. The action can be a touch on a UI component, a scrolling of the screen, a back navigation, etc. Figure 4 shows different situations of usage scenarios with a popular app, Tencent News client.

**User Transaction Table.** To record these usage scenarios, we use a data structure called User Transaction Table (UTT), as shown in Fig. 5. Each entry of the UTT table is a usage scenario, i.e., a $< activity, action >$ tuple. For fast access, the UTT is designed as a hash table, where the names of the *activity* and *action* in combination formulate the hash index into UTT. Each time the trigger of a user transaction is identified (called *running user transaction*), The *activity* name and *action* name of the transaction are used to calculate a hash id, which




(a) User transactions of different Activities




(b) User transactions of the same Activity

**Fig. 4.** Example of usage scenarios

will be the index to access the hash table, then a comparison with the *activity* and *action* fields in the corresponding UTT entry is needed for verification. If it is a hit, then the three fields representing the most recent user transactions of this usage scenario are consulted.

Each recent user transaction field contains the normalized CPU load (as defined in Sect. 3.1) for the respective user transaction. This normalized CPU load was calculated by reading the CPU utilization data and the CPU frequency recorded by Linux/Android when the transaction was detected. We use normalized CPU load for ease of comparison among different transactions. When the three transaction fields are consulted, the maximum of them is taken for predicting the computational requirement of current transaction. Note this is a relatively conservative policy to reduce the risk of underestimation, but it is much better than blindly taking $f_{max}$ of the CPU as the capped frequency for executing current user transaction. Then the capped frequency $f_c$ to be used for the running user transaction can be calculated by following the formula in the definition of the normalized CPU load in Sect. 3.1: $max(U_{norm1}, U_{norm2}, U_{norm3}) = 0.8 * (f_c/f_{max})$, and we will have $f_c = max(U_{norm1}, U_{norm2}, U_{norm3}) * f_{max}/0.8$. Here 0.8 instead of 1.0 is taken to be the target CPU load for the running user transaction, as a CPU load of 1.0 means that the CPU might be too busy to meet the computational requirement, and 0.8 is a high, but safe CPU load for getting a low capped frequency.

**The Framework.** Based on the idea of UTT, the UH-DVFS framework is presented in Fig. 6. It consists of the application layer and the system layer. We design a background service running at the application layer, which has two basic functionalities: dynamic instrumentation and resource monitoring. To capture the user input and the corresponding foreground activity for accessing the corresponding UTT entry, a dynamic instrumentation module is designed,

**User Transaction Table**

| Usage Scenario | | Three Recent User Transactions | | |
|---|---|---|---|---|
| **Activity** | **Action** | $U_{norm1}$ | $U_{norm2}$ | $U_{norm3}$ |
| Activity1 | Tran1(click) | 0.6 | 0.55 | 0.58 |
| | Tran2(touch slide) | 0.41 | 0.45 | 0.43 |
| | Tran3(press back) | 0.33 | 0.35 | 0.31 |
| Activity2 | Tran4(click) | 0.37 | 0.38 | 0.4 |
| | Tran5(touch slide) | 0.55 | 0.50 | 0.52 |
| ..... | | | | |

**Fig. 5.** User transaction history table

which contains the code that we want to inject at runtime when the foreground application calls the API of Android Framework. The resource monitor samples the CPU load as well as the CPU frequency at 8 Hz through the file system in the kernel, and their product is normalized with respect to the max frequency. The UH-DVFS adaptor at the system layer is the key component of the framework. Its user transaction monitor captures the data of user transaction, including Activity information and user's UI manipulation events, which will be formulated as an index into the UTT for consulting its execution history. Upon a hit, the capped frequency for current running user transaction can be calcuated using the formulate described above. Otherwise, the conservative Ondemand Governor of Android will take responsibility, which will set the capped CPU frequency directly up to $f_{max}$. Eventually, the capped frequency $f_c$ will be enforced on the CPU by the DVFS library and the underlying driver offered by Android kernel.

### 3.4 Dynamic Instrumentation

As most modules in the framework depends on the dynamic instrumentation, we elaborate on how it works. First of all, its goal is to provide facilities to capture the foreground Activity information and the user manipulation events on that Activity. In addition, we need to detect the display update to determine the user transaction interval. Figure 7 presents the overview of the instrumentation framework. With it in place, the original application will callback the instrumentation framework to get user transaction data when a call to the Android framework API for running this transaction is triggered. From the user's perspective, it appears that each time a user transaction is a launched, it simultaneously produces an output of user transaction data automatically. This instrumentation hardly has any negative effect on normal running of the original applications. It

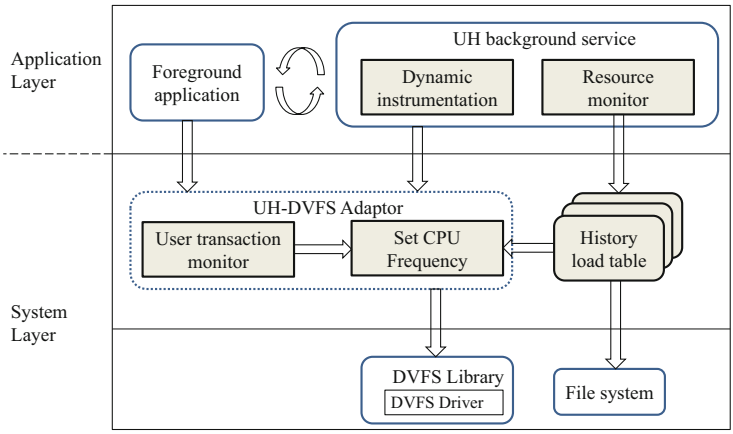



**Fig. 6.** The framework of UH-DVFS

simply hooks methods to the Android framework at runtime. We make use of the open source project named Xposed [3] for this purpose. In Android, Zygote is the root Java process, which is responsible for forking other Java processes. When the framework is installed, an extended app_process is copied to /system/bin. This extended startup process adds an additional jar to the classpath and calls methods from there at certain places. So we can hook or replace the method that we want to monitor and inject our own methods. We illustrate how the respective data are captured when a user transaction is triggered.

**UI Operation Events.** When the user interacts with the foreground activity, the UI touch events are dispatched by the method *dispatchTouchEvent* of the Activity class. Therefore, we hook this method to get motion events of the user interaction by which we can distinguish user operations between "clicking" and "sliding" on the activity.

**UI Upate Events.** Android system updates UI using a message queue model. Messages are placed in a queue and the UI thread loops indefinitely, processing messages from the head of the message queue. Therefore, we can detect UI update events by monitoring the message queue of the main thread, and catch the *invalidate()* invocations at the same time.

**Foreground Activity Information.** We need this information to identify a user transaction. Usually, an Android app consists of a set of Activities, and each one has its own life cycle. When an Activity comes into the foregound, it stays at the running state to receive user interactions. The entry point of this state transition is the *onResume()* invocation of Activity class in the Android framework. Therefore, we can inject our code at this place ,such that when any Activity invokes this API function, we can detect this Activity switching, and get the current foreground Activity in time.

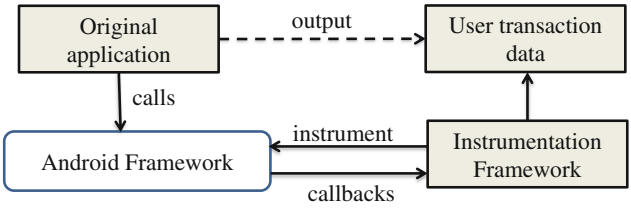



**Fig. 7.** Overview of instrumentation

## 4 Experiments and Results

To validate the effectiveness, we have implemented the framework on real smartphone devices, and use a set of popular Android apps for evaluation.

### 4.1 Experimental Setup

Our experiments are conducted on a Xiaomi-2S smartphone running Android 4.1 OS. It is equipped with Qualcomm Snapdragon APQ8064 CPU, which supports 14 different frequency levels. we measure the energy consumption of Xiaomi-2s with the Power Monitor of Monsoon solutions [2], which samples power data at a rate of 5 KHz. The experiment setup is shown in Fig. 8. The phone is connected to both the power monitor and a laptop running Windows via USB at the same time. We disable the phone's USB battery charge to avoid any sources of inaccuracies in the experiment. With this experimental setup, we measure the energy consumption under the UH-DVFS framework and the default Android DVFS governor respectively for comparison on energy efficiency. To avoid user-introduced differences, we automate the test processes by using the Monkeyrunner [1] tool, which can record and replay the same experimental scenes for UH-DVFS and the default DVFS governor automatically without manual intervention.

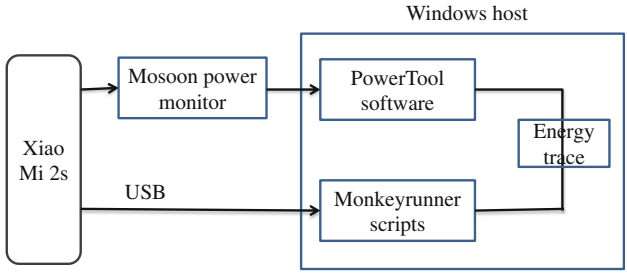



**Fig. 8.** Experimental setup

### 4.2 Energy Saving and Performance Evaluation

In this part, we evaluate the effectiveness of the proposed UH-DVFS scheme on energy savings with a set of interactive apps. The energy savings for different apps are presented in Fig. 9. It shows that our UH-DVFS approach can save the smartphone energy from 10 % to 36 % over the default Governor policy of Android. Note that this result has excluded the energy consumption of the screen, as our objective is for CPU energy reduction. We achieve this by first keeping the phone in a totally idle state with its screen on, and get the energy consumption of the screen. Then we turn the screen off, and perform the automated experiments described earlier. After obtaining the energy data, the earlier tested screen energy consumption is deducted from the overall energy consumption, and the rest amount of energy is used for comparison. Figure 9 also shows variations of energy savings for different application scenarios, because different apps have different CPU load. Usually the lower the actual application performance requirement, the higher the energy saving by our framework. This perfectly matches our objective on this work: avoid consuming unnecessary energy for application scenarios that do not need a powerful CPU setting.

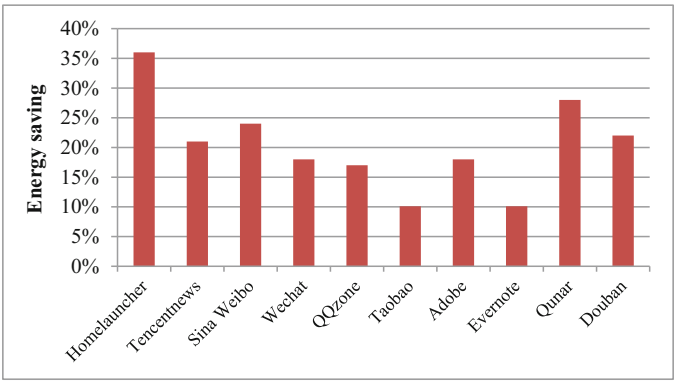
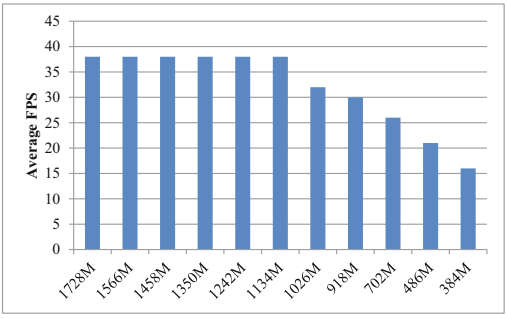


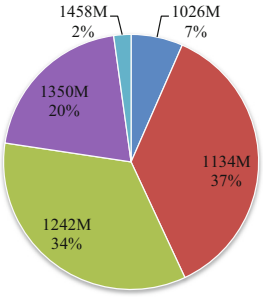

**Fig. 9.** Energy savings of different applications

There is one concern to be addressed: does UH-DVFS achieve energy reduction at the cost of poor performance? Or in other words, is the potential slowdown with a relatively lower capped CPU frequency sacrifing user experiences? To answer this question, we use a commonly used metric, the average frame rates (FPS,Frames Per Second) to decide whether the user experience on a user transaction is satisfied. We measured the instant frame rates using the tool Adreno Profiler offered by Qualcomm. Adreno Profiler can capture the real-time FPS whenever the phone's screen update is detected by connecting to the smartphone with adb command. We measured the average FPS by repeating the same user transaction at least ten times. Our UH-DVFS framework dynamically adjusts the capped frequency of user transactions. We take the Tencent News app as

an example to study the relationship between the average FPS and the capped frequency. In the experiment, we calculate the average FPS of user transactions on Tencent News app's *MainActivity* with different capped frequencies, and the result is presented in Fig. 10. From Fig. 10(a), we can see that the average FPS begins to drop when the capped frequency is lower than 1134 M, which means the performance of user experience can not be satisfied with frequencies lower than this threshold. All frequencies higher than or equal to 1026 M make no negative impact on the quality of user experience for this usage scenario. Figure 10(b) shows the distribution of capped frequencies when Tencent News app is executed under UH-DVFS for a duration of 10 min. The result shows that the portion of capped frequencies lower than or equal to 1026 M only accounts for 7 %, which indicates that the energy saving by UH-DVSF is indeed achieved without sacrificing user experience in most cases.




(a) Average FPS of the same user transaction with different cap frequencies




(b) Cap frequencies distribution with UTA-DVFS scheme

**Fig. 10.** FPS and Cap frequency measurement on the user transaction of Tencent news application

To further get an idea on how serious the average FPS is affected by CPU frequencies, we measured the average FPS under different frequencies for all the apps used earlier. The result is plotted in Fig. 11. It can be observed that the threshold frequencies meeting the quality of user experiences (e.g., 30 FPS) for different user transactions are also different, which means an adaptive DVFS scaling for different scenarios is needed.

We have measured the extra CPU overheads incurred by the UH-DVFS scheme, as it only add hooks at very few places, such as the triggering of a user transaction, it incurs very little performance overhead. According to our measurement, the CPU load of UH-DVFS is only around 1 % during an active user transaction, and it is nil during the period without user input.

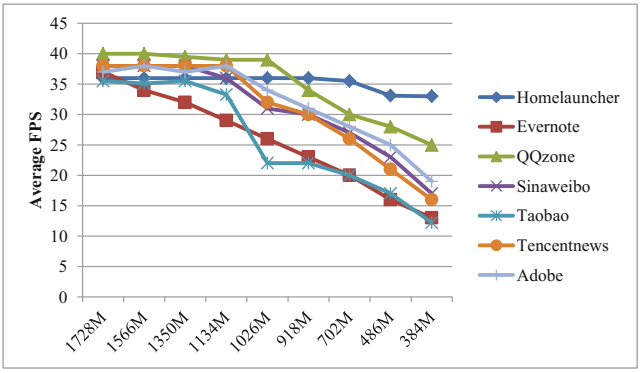



**Fig. 11.** Average FPS of different user transactions

## 5 Conclusions

Battery life is an important concern for smartphone users. Existing DVFS-based techniques for CPU energy reduction are either not responsive (sacrificing user experience), or too conservative (wasting batter power). In this paper, we propose UH-DVFS, a novel usage history-directed DVFS framework to save energy for interactive operations, which are a primary source of smartphone energy consumption. UH-DVFS is motivated by an important observation that the repetitive occurrences of the same usage scenario usually have quite stable and predictable performance behaviors. By leveraging this observation with a User Transaction Table (UTT), we are able to set the capped frequency for recurring user transactions to a lower frequency level without sacrificing user experiences. We have evaluated the effectiveness of UH-DVFS with real-world applications. Under UH-DVFS, the smartphone XiaoMi-2s achieves considerable energy reduction (ranging from 10 % to 36 %) compared to the default DVFS governor of Android. In future research, we will take account of the GPU and network together, as GPU is also a major energy-consuming component, and we believe that the integrated CPU-GPU DVFS is a new direction to study further. In addition, we will study the user transactions with more factors to refine the model, such that the refined user transactions are more stable than our current model in terms of performance.

**Acknowledgments.** This work is supported by the grant of Shenzhen municipal government for basic research on Information Technologies (No. JCYJ20130331144 751105).

## References


1. Android monkeyrunner. http://developer.android.com/tools/help/MonkeyRunner.html
2. Monsoon power monitor. https://www.msoon.com/
3. Xposed. https://github.com/rovo89/XposedBridge
4. Bai, Y.: Memory characterization to analyze and predict multimedia performance and power in an application processor. Marvell White Paper (2011)
5. Chang, Y.-M., Hsiu, P.-C., Chang, Y.-H., Chang, C.-W.: A resource-driven dvfs scheme for smart handheld devices. ACM Trans. Embed. Comput. Syst. (TECS) **13**(3), 53 (2013)
6. Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., Estrin, D.: Diversity in smartphone usage. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, pp. 179–194, ACM (2010)
7. Kang, J.-M., Seo, S.s., Hong, J.W.-K.: Usage pattern analysis of smartphones. In: 2011, 13th Asia-Pacific Network Operations and Management Symposium (APNOMS), pp. 1–8, September 2011
8. Kim, S., Kim, H., Hwang, J., Lee, J., Seo, E.: An event-driven power management scheme for mobile consumer electronics. IEEE Trans. Consum. Electron. **59**(1), 259–266 (2013)
9. Le Sueur, E., Heiser, G.: Dynamic voltage and frequency scaling: the laws of diminishing returns. In: Proceedings of the 2010 International Conference on Power Aware Computing and Systems, pp. 1–8. USENIX Association (2010)
10. Pallipadi, V., Starikovskiy, A.: The ondemand governor. In: Proceedings of the Linux Symposium, vol. 2, pp. 215–230, sn (2006)
11. Pathak, A., Hu, Y.C., Zhang, M.: Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 29–42, ACM (2012)
12. Pathak, A., Hu, Y.C., Zhang, M., Bahl, P., Wang, Y.-M.: Fine-grained power modeling for smartphones using system call tracing. In: Proceedings of the Sixth Conference on Computer Systems, pp. 153–168, ACM (2011)
13. Pathania, A., Jiao, Q., Prakash, A., Mitra, T.: Integrated cpu-gpu power management for 3d mobile games. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6, IEEE (2014)
14. Shye, A., Scholbrock, B., Memik, G.: Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 168–178, ACM (2009)
15. Song, W., Sung, N., Chun, B.-G., Kim, J.: Reducing energy consumption of smartphones using user-perceived response time analysis. In: Proceedings of the 15th Workshop on Mobile Computing Systems and Applications, pp. 20, ACM (2014)
16. Zhang, L., Bild, D.R., Dick, R.P., Mao, Z.M., Dinda, P.: Panappticon: event-based tracing to measure mobile application and platform performance. In: 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), pp. 1–10, IEEE (2013)
17. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 105–114, ACM (2010)