2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

PIFA: An Intelligent Phase Identification and Frequency Adjustment Framework for Time-Sensitive Mobile Computing

Xia Zhang[1], Xusheng Xiao[2], Liang He[3] Yun Ma[4], Yangyang Huang[4], Xuanzhe Liu[4], Wenyao Xu[5], Cong Liu[1]
[1]University of Texas-Dallas, [2]Case Western Reserve University, [3]University of Colorado-Denver, [4]Peking University, [5]University at Buffalo

***Abstract*—Due to the limited battery capacity of mobile devices, various CPU power governors and dynamic frequency adjustment schemes have been proposed to reduce CPU energy consumption. However, most such schemes are app-oblivious, ignoring an important fact that real-world applications often exhibit multiple execution phases that perform different functionality and may request different amounts of hardware resources. Having a unified app-level frequency setting for different phases of an application may not be energy efficient enough and may even violate the desirable latency performance required by certain phases. Motivated by this observation, in this paper, we present PIFA, which is an intelligent Phase Identification and Frequency Adjustment framework for energy-efficient and time-sensitive mobile computing. PIFA addresses two major challenges of fully automatically identifying different execution phases of an application and efficiently integrating the phase identification results for runtime frequency adjustment. We have fully implemented PIFA on the Android platform. An extensive set of experiments using real-world Android applications from multiple app categories demonstrate that PIFA achieves closely better performance than the desired latency requirement specified for each phase, while dramatically reducing energy consumption (e.g., $>$30% energy reduction for most apps) and incurring rather small runtime overhead (e.g., $<$5% overhead for most apps).**

I. Introduction

Due to the limited battery capacity of time-sensitive mobile devices, energy optimization has been an important research thesis for mobile computing. As reported by recent studies of user activity, it is most effective to develop power optimization techniques focusing on the CPU and screen of mobile devices [35]. Dynamic voltage frequency scaling (DVFS) is a key technique that reduces CPU energy consumption through dynamically adjusting the supply voltage and operating frequency. We find out that the energy efficiency of most existing DVFS schemes and CPU power governors developed for mobile devices can be further improved, as such schemes are app-oblivious and mainly focus on exploring the tradeoff between raw performance and energy efficiency while ignoring the following important facts.

First, the CPU processing capacity available on most today's mobile devices (e.g., smartphones) far exceeds the maximum hardware requirements of many applications, which may cause the latency performance of such applications not to improve after setting the CPU's frequency above an application-dependent threshold (see the motivational measurements-based case studies in Sec. II). Moreover, real-world applications often exhibit multiple execution phases (e.g., a game application often has two phases: a menu operation phase and an actual gaming phase). Applications in different phases perform different functionality, request different amount of hardware resources, and even have different latency performance metrics (e.g., response times for menu operations and frame-per-second (FPS) for actual gaming). Thus, defining a unified and app-oblivious DVFS setting for multi-phase applications is clearly not energy efficient enough and may even violate the desired latency performance required by certain phases. Furthermore, for most user-interactive applications such as mobile games or video-based applications, the upper limit on an application's latency performance is often determined by human perceptual abilities. Achieving better performance than this upper limit by supplying the maximum CPU frequency is unnecessary but results in greater energy consumption.

Motivated by these observations, this paper seeks to develop an application-stateful power management framework for mobile computing, with the goal of identifying the ideal core frequency setting for running an application that yields the most desirable performance while significantly reducing energy consumption. We present PIFA—an intelligent Phase Identification and Frequency Adjustment framework for time-sensitive mobile computing. There are two major challenges addressed by PIFA: (*a*) *how do we automatically and accurately identify the execution phases*, and (*b*) *how can we efficiently integrate the phase identification results for frequency adjustment*. To address challenge (*a*), recent studies find that applications usually request similar amount of hardware resources within the same phase, and different amounts of hardware resources in different phases [23], which can be precisely captured by clustering analysis [17], [20]. However, such *knowledge discovery process* requires a certain amount of data and is too expensive to include in runtime decision making (i.e., challenge (*b*)).

To address these challenges, PIFA employs a synergy approach that combines offline analysis and online analysis: (1) offline analysis is used to perform expensive analysis (clustering analysis for phase identification) and the results are summarized as a classification model (a phase classifier) that can be repetitively and efficiently applied in future analysis; (2) online analysis is used to perform light-weight decision making according to the monitored resource usage information on CPUs, GPUs and memories and adjust CPU frequency accordingly; (3) the intelligence of the online analysis is

Work supported by NSF grant CNS CAREER 1750263.

978-1-7281-0678-6/19/$31.00 ©2019 IEEE
DOI 10.1109/RTAS.2019.00013

enabled by leveraging the classification model obtained from the offline analysis. Such synergy of offline and online analysis not only achieves high accuracy in identifying phases, but also greatly reduces the runtime overhead on decision makings.

With the phase identification, PIFA can then adjust the voltage and frequency of the core on which an application is scheduled to run according to the "sweet frequency" setting that can achieve the desired performance of the corresponding phase. PIFA defines the sweet frequency for each phase of an application by exploiting two facts discussed above, which uses the maximum of the following two values: (*i*) the frequency threshold, where further increasing frequency above this threshold does not further increase performance (or increases performance in a negligible manner), and (*ii*) the frequency value that achieves the upper limit of the performance due to human perception limits (if any).

We have implemented PIFA on the Android platform. PIFA is fully automated and thus it is practical and scalable to a large number of real-world Android apps.[1] Specifically, in offline analysis, we leverage DroidWalker [25] to automatically generate the events for identifying different phases of an app. DroidWalker is a systematic app-exploration tool that is shown to achieve comparable or even better performance in exposing various behaviors in Android apps than the state-of-the-art app-exploration tools [13]. The clustering analysis is then applied on the collected resource utilization data, where data points within a cluster form a phase, and a subsequent analysis based on the identified phases is used to train a phase classifier.

The output of offline analysis for an app is a small model file (usually 30-50 KB) that can be embedded in the app. The online analysis leverages this model file to perform application-stateful power management, incurring rather small runtime overhead. Due to this automatic process of phase identification and the light-weight runtime analysis, our implementation can easily scale w.r.t. the number of apps. Since DroidWalker can replay events on apps across different device models, the offline analysis can be applied to train phase classifiers for a series of different device models. Each phase classifier will be leveraged in the online analysis on the corresponding model. As DroidWalker can replay the generated test cases in other devices, this process can be fully automated in other devices before app installation, i.e., enabling auto tuning. This is also the fundamental reason why our design of PIFA features portability and auto-adaptation for a wider range of mobile devices. To the best of our knowledge, no prior works on either DVFS or clustering provides such a synergistic, automated, and scalable solution.

We have conducted an extensive set of experiments using 18 real-world Android apps, where 13 apps are downloaded from official Google Play (e.g., Angry Birds and WeChat) and 5 apps are obtained from open-source app market (e.g., F-Droid [2]), which belong to three common categories: UI apps that often desire real-time performance, game apps and video apps that desire certain FPS performance. The evaluation results show that PIFA is rather effective in reducing energy consumption (e.g., >30% energy reduction for most apps) while maintaining desired latency performance level in each phase of an application execution, particularly for more resource-demanding apps (e.g., game and video apps). The overhead incurred by PIFA is reasonably small under all scenarios (e.g., <5% overhead for most apps), which does not offset the energy saved by PIFA. Moreover, the phase management component is proved to be the key for PIFA to reduce energy consumption while achieving desirable performance, as defining a unified sweet frequency for all phases of an application either yields more energy consumption or violates the desirable latency performance required by certain phases.

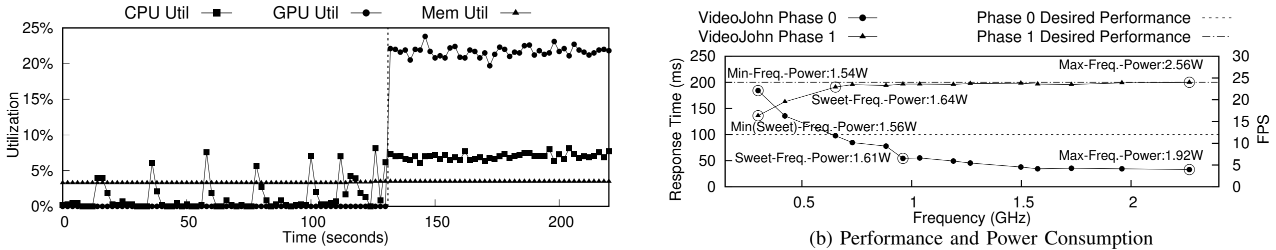
II. Motivation and Background

To motivate our design philosophy behind PIFA, we perform a set of measurement-based case studies using three popular Android apps that represent three common categories of mobile apps: the "Candy Crush" app representing game apps where the performance metric is FPS, the "VideoJohn" app representing video apps where the performance metric is also FPS, and the "Scale View" app representing UI apps where the performance metric is timing (i.e., response time). Through these case studies, we seek to answer the following questions: (*i*) is there an application-dependent frequency threshold? (*ii*) whether setting the maximum CPU frequency will yield high power consumption but performance is unnecessarily higher than the desired performance either specified by system designers or due to human perceptual ability limit? (*iii*) are there multiple phases existing in an app that may require dramatically different resources and have different criteria on defining desired performance?

We perform the case studies on an Android smartphone Nexus 5 equipped with a Qualcomm 4-core processor that supports 14 different frequencies ranging from 0.3GHz to 2.26GHz. For each app, we measure the performance of running each app for 3 minutes by using Monkey [5] to automatically generate the events for exposing various behaviors of the app. Experiments are performed under various CPU frequency settings, from the lowest frequency value 300 MHz to the maximum value 2.26GHz. We also use a popular profiling tool Trepn [6] to monitor runtime resource utilization information on CPU, GPU, and memory under a fixed frequency setting of 2.26GHz.

The video app: "VideoJohn" is a video app that allows users to play and edit videos. As seen in Fig. 1a, the resource utilization on all three types of resources can be considered to experience two observably different stages (relatively low GPU resource utilization within [0, 130s] and high GPU resource utilization during [130s, 220s]). According to the incurred operations, these two stages reflect two execution phases available in this app: a video editing phase yielding low resource utilization and a video play phase yielding high resource utilization. The performance metrics for these two phases are also different: the editing phase (phase 0) is a UI

[1] Apps are generally used to denote mobile applications.



(a) Resource utilization

(b) Performance and Power Consumption

Fig. 1: Measurements of VideoJohn

phase with the response time metric and the video play phase (phase 1) is a video phase with the FPS metric.

Fig. 1b shows the measured performance and the desired performance for these two phases under various frequency settings. The desired performance for the editing (video play) phase is set to be 100 ms (24 FPS) due to human perceptual ability limit [27] (the app specification given by the developer). For the video play phase, most of the frequency settings (i.e., those above 0.65 GHz) can yield closely the same performance as the desired performance of 24 FPS. This is because most today's Android smartphones have specific hardware for frame-decoding. Thus, the video frames are not processed by CPUs and CPUs are only responsible for few simple management work. As the frequency threshold is also 0.65 GHz for the video play phase, the sweet frequency is set to be 0.65 GHz as it yields the lowest power consumption. For the video editing phase, we can see that a frequency of 0.65 GHz can ensure a performance of 100 ms while a frequency threshold is identified to be at 0.96 GHz. Thus, setting the sweet frequency to be at 0.96 GHz can ensure the desired performance while achieving a much lower power consumption compared to the maximum frequency setting.

Summary: From the case studies, we obtain the answers for the earlier questions that motivate the design of PIFA. First, (potentially) for many real-world applications, there is an application-specific frequency threshold (on a specific mobile device). Second, setting the maximum CPU frequency often yields significant power consumption and performance that is unnecessarily high. Third, many real-world applications have multiple execution phases due to various embedded functionality, which often incur observably different resource utilization profiles on CPUs, GPUs, and memory. This suggests that it may be possible to detect execution phases based on the runtime resource utilization information. Fourth, it may be judicious to set the sweet frequency for executing a specific phase to be the maximum value between the frequency threshold and the minimum frequency that achieves the the desired performance for that phase. Intuitively, doing so can ensure desired performance to be reached even if interference on an application's execution exists such as environmental noise.

III. System Design of PIFA

Figure 2 shows the overview of PIFA. PIFA consists of both *offline* and *online* analyses. In the offline analysis, PIFA profiles an app to identify its number of phases and the corresponding sweet frequency for each phase. PIFA also trains a phase classifier based on the profiling data, which is used later in the online analysis. In the online analysis, PIFA schedules an app to execute in an idle CPU core whenever possible, and intelligently adjusts the CPU frequency to the sweet frequency based on which phase the app is in [2]. The online analysis uses the classifier trained in the offline analysis to identify phases during runtime. We next provide the details of the offline and online analyses of PIFA.

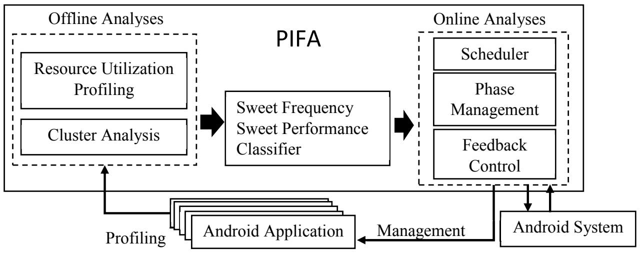


Fig. 2: Overview of PIFA

A. Offline Multi-Phase Identification

Offline analysis aims to capture the major energy consumption patterns (phases) exhibited by an app and train a phase classifier that can dynamically identify phases for online analysis. Section II shows that resource utilization is strongly correlated with the phases in apps. More specifically, given two data points of resource utilization, if they are in different phases, their values should be quite different, while their values should be similar otherwise. Such data characteristics can be precisely captured by using typical cluster analysis methods such as K-Means [17], [20]. Thus, the offline analysis profiles the app to collect its resource utilization, and leverages the cluster analysis to identify the major phases. We describe these two steps in details as follows.

Resource Utilization Profiling: Given an app, the offline analysis runs the app and performs a series of automatic interactions with the app to obtain its resource utilization at different phases. In general, if an app is released with descriptions of its major functionality and corresponding functional

[2] For most apps, the major functionality that is most valuable and perceptible to users usually runs in a single thread (e.g., such as videos and games in the UI thread), and thus PIFA focuses on optimizing the frequency value of the core that runs the major functionality of an app.

tests, we can run these tests to exercise the major functionality and profile its phases. However, apps released in App Stores such as Google Play [3] may not come with functional tests, and it is difficult to determine all the major functionality from the app descriptions since these descriptions focus on describing only the features that are attractive to users.

To explore apps' functionality, we leverage DroidWalker [25], an app-exploration tool to automatically generate the events for exposing behaviors of the app. DroidWalker can systematically explore the app in a depth-first search strategy by triggering UI events including clicking, keyboard stroking, and scrolling. Compared with other state-of-the-art Android testing tools, DroidWalker can achieve comparable or even better code and activity coverages for Android apps. In addition, for each explored location, DroidWalker generates a test case that can be executed to reproduce the corresponding app behaviors across different device models.

The *reproducibility feature* of DroidWalker makes it very useful for identifying the sweet frequency for each phase, which we will discuss later. We run every app using DroidWalker to systematically explore the apps' behaviors. The exploration continues until the activity coverage reported by DroidWalker does not increase in 5 minutes (which is long enough and expected to be reasonable in practice). Every app is launched and explored for three times. In this way, we can collect enough data points to obtain more accurate clustering results.

At runtime, three types of resource-utilization data are collected: CPU, GPU and memory. Based on the findings in Section II, it is quite effective by applying clustering analysis to identify phases. Note that for many cases it may not be accurate enough to simply distinguish phases by using only CPU utilization. It is common that two phases of an app may have similar CPU utilizations but very different GPU and memory utilizations. We thus use all three measures for phase identification, which yields higher accuracy.

Cluster Analysis: To identify phases of the app, the offline analysis applies the clustering analysis on the collected resource utilization data. With such clustering process, data points with similar values are clustered together to form a phase.

In general, an app has a few major phases and may contain various minor phases. As most of the minor phases are transient ones that exist for a short time, such phases are not interesting for PIFA to use for saving energy. For the reason of practicality, it is not desirable for the minor phases to add too much complexity on the phase identification. Therefore, we use K-means clustering algorithm, a type of unsupervised learning algorithms whose k values can be configured to a small number. This algorithm can cluster the resource utilization data into k (k is a small number) clusters $\{c_1, c_2, ..., c_k\}$ for identifying major phases. It has been shown empirically in both prior work [24] and our extensive empirical study presented in Sec. V that the majority of apps have a few major clusters, while the minor phases can be ignored. Thus, K-means with K being a small number is suitable for our setting. Note that other unsupervised learning algorithms can also be used instead of K-means.

To obtain the optimal value of k, we apply K-means with $k = 1, 2, \ldots, 6$. We choose the k ranging from 1 to 6 because based on our empirical observations, where the number of phases is not larger than 6 for most Android apps. Such an observation is also consistent with the previous report [23]. We compute the average error rate for every single value of k and choose the value that achieves the lowest average error rate. However, if two adjacent values of k are quite similar, we choose the smaller value since phases with similar values often have very similar sweet frequencies. Thus, for each k, we also compute the delta for the error rate of k and $k + 1$. If the delta is smaller than 0.05, then k should be chosen. Since in this case, increasing the k value does not reduce the error rate significantly but could result in more clusters, which makes the analysis unnecessarily complicated.

The complexity of the clustering method is $O(n * k * i)$, where n is the number of input resource utilization data, k is the number of clusters and i is the number of iterations. Obviously, such complexity is prohibitively high for runtime decision making, but is reasonable when running offline (e.g., $<$ 5 seconds for a desktop to find clusters in our experiments).

The clustering analysis identifies k major phases of the app. Based on the result, we label the data points with k phases, and use the labelled data to train the classifier. In the online analysis, we choose the distance-based classification as it works better with the major phases identified using K-means. For each phase, we choose the corresponding app behaviors and use DroidWalker to reproduce them. At the same time, we adjust the CPU frequency and measure the sweet frequency using the preceding approach described in Section II.

B. Online Frequency Adjustment

The goal of the online analysis is to schedule the execution of apps to CPU cores, and adjust their frequencies based on the performance and resource utilization data collected at runtime. Fig. 3 shows the overview of the online analysis of PIFA, which consists of three main components: scheduler, phase management, and feedback control. The scheduler component is responsible for assigning an app to a CPU core according to some criteria. The scheduling is performed whenever a core is available and at least one app is waiting to be executed. The phase management component collects the resource utilization data and identifies which phase the app is running in. Based on the phase, it assigns the sweet frequency to the corresponding CPU core. The feedback controller component collects the performance data and adjust the CPU frequency accordingly if the performance deviates from the desired performance.

Scheduler Component: The scheduler component includes two queues and one scheduler. Apps to be executed are put into two different queues: Q_f is for foreground apps and Q_b is for background apps. Whenever there are cores available, the scheduler chooses an app from the queues to start. In Android system, two foreground apps running concurrently are not allowed. But several background services can run concurrently.

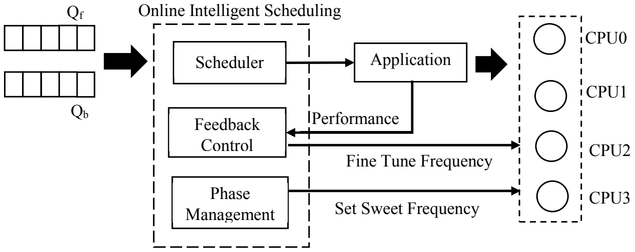


Fig. 3: Overview of online analysis

Also, one foreground app and several background apps can run concurrently. Thus, the scheduler component does not schedule two foreground apps to start at the same time, and the services are scheduled to execute if there are cores available. If there is an app running at the foreground, the scheduler checks Q_b directly. If not, the priority is given to Q_f; if Q_f is empty, then Q_b is checked.

Phase-Management Component: After assigning the app to a CPU core, PIFA also decides the frequency at which the app should run. Since an app may include multiple phases and each phase has its own frequency, the phase-management component first identifies the phase that the app currently resides in. To this end, the classifier trained in the offline analysis is used to classify the resource-utilization data collected during runtime. Based on the identified phase, the phase-management component sets the CPU to the corresponding sweet frequency.

Algorithm 1 shows the detailed steps performed by the phase management component. First the resource utilization data is obtained through reading system profiling files (Line 1). The phase classifier is then applied on the resource utilization data (Line 2). The classifier classifies the resource utilization data by computing the distances between the data and the k centroids, and associating the data to the cluster with the minimal distance. To tolerate potential background noise, we consider that the phase has been changed if a new classification result appeared at least $Switch\_Treshold$ times in a row (Lines 3-8).

Algorithm 1 Phase management Component

```
1: $(U_{cpu}, U_{gpu}, U_{mem}) = ReadSystemFiles()$
2: $Current\_Phase = Classifier(U_{cpu}, U_{gpu}, U_{mem})$
3: if $Current\_Phase\ = Last\_Phase$ then
4:     $Count \leftarrow Count + 1$;
5: else
6:     $Count \leftarrow 0$
7: if $Count > Switch\_Threshold$ then
8:     $Freq \leftarrow Current\_Phase.Freq_s$
9: $Last\_Phase \leftarrow Current\_Phase$
```

Feedback-Control Component: The phase-management component enables PIFA to quickly and accurately identify the sweet frequency for every phase. We further apply the feedback-control component to deal with (often small) runtime performance deviations. The feedback-control component dynamically adjusts the CPU frequencies based on the collected performance data. The workload in Android OS always fluctuates and is difficult to predict, which could have severe impacts on the performance of running apps. Therefore, when the system workload is high, the performance might deviate significantly from the desired performance with CPU frequency set as the sweet frequency. To mitigate this issue during heavy workload periods, the feedback-control component needs to make online analysis. The feedback-control component measures the deviation of the current performance from the desired performance, and adjusts the frequency according to the gap between the current performance and the desired performance.

The feedback-control component reads the performance data from the system profiling files, which is updated through online profiling. We compute the average performance in the past 5 seconds as the measured performance to mitigate runtime errors, and compare the average performance with the desired performance: if the average value is $\alpha\%$ lower than the desired performance, the CPU frequency is increase by a level; if the average value is $\beta\%$ higher than the desired performance, the CPU frequency is decreased by a level. α and β are derived from the positive and negative deviations of the sweet performance (plus 10% for measurement errors).

IV. System Implementation

PIFA is implemented as a set of Android tests, a Java program, a set of Android services that schedule execution and manage phases. Root permissions are required to read/write the system files of frequencies and power governors. The implemented PIFA is full-automated and can be easily applied by users, which is not device- and/or app-dependent but is rather easily scalable to a large number of apps and devices.

The offline analysis of PIFA is implemented as a set of automatically generated Android tests that interact with apps to be executed and collect the profiling data. The machine learning algorithm is implemented as a Java program based on the library provided by Weka [15]. The output of the offline analysis are the identified phases with their sweet frequencies and sweet performances, and a classifier model file that can be used for online classification. For each phase the sweet frequencies and sweet performance take up 6 bytes (4 bytes for frequency and 2 bytes for performance), and the size of classifier model files ranges from 30KB to 50KB. Given that most apps in the market range from 10-50M, it is very light weight to embed model files in apps (<1% of the app binary size). For a given app of a specific version and a given device model, we need to perform the offline analysis only once, and the output of the offline analysis can be attached as a part of the app. As the model files are small and the offline analysis is fully automated, PIFA can be applied to various device models and easily generate model files through offline analysis for them. Therefore, the offline analysis can be performed by either developers or app markets, and the results can be embedded in the app to be released in the app markets. In other cases when the model files are not available for the model of the target device, PIFA can automatically run the offline analysis for the target model and generate the model files.

The online analysis of PIFA is implemented as a set of Android services and a timer thread. The apps to be executed are first instrumented to report their execution status and performance data as files in a specific folder. The scheduler and feedback control components read these files periodically for decision makings.

App Instrumentation: To allow PIFA to take proper control of apps, apps need to be instrumented to achieve two fined-grained controls: affinity setting and performance collecting. For open-source apps, our instrumentation is done at the source code. Since most Android apps on the market are not open-source, we also provide techniques to directly instrument app binaries (i.e., apk files). To instrument app binaries, we leverage Soot [36], [9] to systematically convert Android's Dalvik bytecode into Jimple, perform code transformations on Jimple, and then convert the Jimple representation back to Dalvik bytecode.

V. Evaluation

In our evaluations, we seek to answer the following research questions:

- **RQ1**: How effective is PIFA in identifying the performance sweetspots (i.e., saving energy while maintaining desired performance level)?
- **RQ2**: How efficient is PIFA in controlling apps to reach the performance sweetspots (i.e., saving energy while introducing low overhead)?
- **RQ3**: How does the phase management contribute to the effectiveness of PIFA in identifying performance sweetspots?
- **RQ4**: How could PIFA be generalized for different phone models?

A. Evaluation Setup

Our evaluations are conducted on an Android smart phone Nexus 5 with Qualcomm Snapdragon 808 CPU, 2GB LPDDR3 memory, and 2700 mA battery. In total, 18 representative apps are used in the evaluations, including 13 popular apps from Google Play (e.g., WeChat, Angry Birds), 2 open-source apps from open-source app market F-Droid [2], and 3 open-source apps from other online sources that provide open-source Android apps. Specifically, we have 3 video apps including KM Player, OpenGL Video, and VideoJohn; 3 game apps including Angry Birds, Candy Crush, and Replica Island; and 10 UI apps including RAR, Desk Clock, Note Pad, CM Security, SD Card Cleaner, Cool Reader, Power Tutor, ScaleView, Alogcat, DidI, WeChat, and Mobike.

We choose these representative apps based on their *popularity*. For the game apps, the download counts of Angry Birds and Candy Crush are around 1,500,000, and the download count for KM Player is around 200,000. The remaining apps are all UI apps and many of them are used by thousands of users. These apps fall in different categories and have different complexities. Specifically, WeChat is one of the most popular apps used all over the world (5,177,501 download counts in Google play and 762 million monthly active users). WeChat has over 600k lines of Java code and 607 distinguished activities [37], representing the state-of-the-art complexity of mobile apps. We believe that the apps used in the evaluation exhibit sufficient practicality, complexity, and variety such that PIFA's effectiveness can be assessed in a comprehensive manner.

After offline clustering, among the 18 apps, 7 apps are multi-phase apps (5 two-phase apps and 2 three-phase apps) and other apps are single-phase apps. For example, in the game Replica Island, the actions in the `launching/selection` UIs are mapped to one phase of menu manipulation, and the `touch/swipe/drag` actions are mapped to another phase of game play. In addition, we find that the more complex an app is, the more phases are identified: WeChat and Mobike (which provides bike-sharing services) are the two apps with three phases. Take WeChat as an example. The general UI actions such as chatting and searching are mapped to one phase; scanning the QR code (which uses the camera to capture QR images) is mapped to the second phase; reading ads or news in the WebView (which involves browser kernels) is mapped to the third phase.

Table I shows the CPU utilization for running each app under the maximum frequency 2.26GHz, and the sweet frequency identified for each identified phase of each app. These sweet frequencies are used for the online frequency adjustment.

Comparison approaches: To demonstrate the effectiveness of PIFA, we evaluate the performance and energy efficiency of six methods: three CPU governors without PIFA and three CPU governors with PIFA. A key novelty of PIFA is the synergistic approach that synthesizes offline analysis and online analysis, allowing high accuracy in phase identification and low runtime overhead on DVFS adjustment. Prior work mostly focuses on the separate consideration of either offline clustering or online DVFS adjustment. To the best of our knowledge, no prior work on either DVFS or clustering techniques can reach this degree of automation and scalability, and thus we chose to compare against native Android power governors. We choose three representative CPU governors:

- Performance governor sets the CPU to the highest frequency and is more energy intensive.
- On-demand governor boosts the CPU frequency to the highest frequency when a workload comes and decreases the frequency gradually when the workload abates. It is less energy intensive.
- Conservative governor promotes the CPU frequency when a larger and more persistent workload is put on CPU, saving more energy with possibilities for choppy performance.

We also compare with PIFA without phase control, which yields the similar performance as existing online DVFS analysis, because they share the same intuition. The results demonstrate the value of our synergy.

Energy Measurement: For each experiment, we use Droid-

TABLE I: CPU utilization under the maximum frequency 2.26GHz and the sweet frequency

Angry Birds(AB): 4.73% / 0.96GHz	Candy Crush(CC): 9.31% / 1.54GHz	Replica Island(RI-P1): 5.17% / 0.65GHz
OpenGL Video(OV): 4.01% / 0.30GHz	KM Player(KM-P0): 8.73% / 0.30GHz	Video John(VJ-P1): 8.45% / 0.65GHz
Desk Clock(DC): 2.13% / 0.73GHz	CM Security(CM-P0): 0.54% / 0.42GHz	CM Security(CM-P1): 8.73% / 0.73GHz
Note Pad(NP): 0.82% / 0.30GHz	RAR: 0.70% / 0.30GHz	SDC Cleaner(SDC-P0): 0.15% / 0.65GHz
SDC Cleaner(SDC-P1): 0.30% / 0.96GHz	KM Player(KM-P1): 0.30% / 0.30GHz	Cool Reader(CR): 0.41% / 0.30GHz
Power Tutor(PT): 4.17% / 0.30GHz	Video John(VJ-P0): 1.75% / 0.96GHz	Replica Island(RI-P0): 0.07% / 0.30GHz
Alogcat(AC): 2.92% / 0.96GHz	Did I(DI): 0.73% / 1.03GHz	SubScale View(SV): 2.91% / 0.88GHz
WeChat(WC-P0): 0.82% / 0.96GHz	WeChat(WC-P1): 3.98% / 1.57GHz	WeChat(WC-P2): 2.64% / 1.26GHz
Mobike(MB-P0): 1.03% / 0.88GHz	Mobike(MB-P1): 2.81% / 0.96GHz	Mobike(MB-P2): 5.43% / 1.26GHz

Walker to automatically generate events for an app, and use a popular energy profiling tool trepn [6] to record the energy consumed during each experiment. For energy consumption, we compare the energy consumed only for running the specific app under each method in the first experiment. Such a step is done by using the measured energy consumption when running the app under each method minus the measured energy consumption when running a “dummy” app (which does nothing) under the same method. We seek to evaluate how much energy can be saved when running each app under PIFA compared to using only Android CPU governors. We also show the device-level energy saving by comparing methods with PIFA and w/o PIFA (Section V-D). In all experiments, the energy measured is the total system energy consumed by all system components.

B. Overall Effectiveness

Fig. 4 shows the evaluation results using boxplot for the 18 evaluated apps under the six methods. For each subfigure, the y axis represents the performance metric and the x axis represents the energy consumed under the corresponding baseline approach (i.e., the original Android CPU governor without applying PIFA). Moreover, the percentage above the whisker of a box plot represents the percentage of the energy savings PIFA combined with the corresponding CPU governor achieves compared to the baseline CPU governor. The predefined desired performance is denoted by the dotted line in each figure.

We obtain the following two major observations by analyzing the data shown in Fig. 4. First, for most game and video apps, PIFA achieves performance close to the desired performance or closely better performance while significantly reducing energy consumption. The three methods associated with PIFA achieve better but close performance to the desired performance, and reduce energy consumption by 10% - 38% (respectively, 40% - 71%) compared to the corresponding baseline method for the three game apps (respective, the three video apps); while the Android CPU governors often achieves unnecessarily better performance while incurring a significantly larger amount of energy consumption. For example, the performance governor yields an absolute energy consumption of 231J for running “Candy Crush”; while “PIFA + Conservative” reduces energy consumption by 231J · 22% = 50.82J for running the same app. The energy savings under video apps are even more significant. This can be explained by jointly considering Fig. 4 and Table I together. As seen in Table I, the sweet frequency identified for video apps is much lower compared to the game apps. This is because the Android phone used in our experiments has a dedicated frame-decoding hardware. Thus, the video frame processing workload does not utilize CPUs intensively. Due to such a low sweet frequency, the energy saving for video apps becomes significant when applying PIFA.

Second, PIFA is more effective in saving energy for game and video apps and less effective in saving energy for UI apps that incur rather low workload. As seen in Table I, the UI app “Did I” incurs a CPU utilization of only 0.73% (since the only thing app performs is to open up a text-based menu description which is a highly optimized Android component). Thus, even if PIFA can effectively reduce the CPU frequency, the resulting energy saving may not be noticeable due to the low CPU workload. Indeed, the absolute energy consumption values for the “Alogcat”, “Did I”, and “ScaleView” UI apps are merely 19.26J and 23.58J, 19.13J and 15.28J, and 16.72J and 18.97J under PIFA and the original CPU governor (both averaged among the three associated methods), respectively. Also, due to the small workload incurred under such apps, the interference caused by measurement or environmental noise may become dominant. This explains why the PIFA-based approaches actually yields more energy consumption for two apps, i.e., -33% and -55% for the “Did I” app under PIFA+Performance and PIFA+Ondemand, respectively, and -18% for the “Cool Reader” app under PIFA+Ondemand (incurring a rather small CPU utilization of 0.41%).

Another interesting observation is that for the three game apps and three video apps with FPS as the performance metric, all six methods achieve similar performance as the specified desired performance. The major reason is that for game and video apps, developers usually maintain a constant FPS to make the code easier for maintenance and save certain CPU cycles. Thus, increasing CPU frequency cannot exceed the performance threshold for these apps; while for the PIFA-based approaches, it demonstrates the accuracy of PIFA in setting the sweet frequency to achieve the desired performance.

Summary for RQ1: In general, PIFA is effective in saving energy while maintaining desired performance levels by accurately identifying the execution phase and setting the corresponding sweet frequency for the apps. PIFA becomes

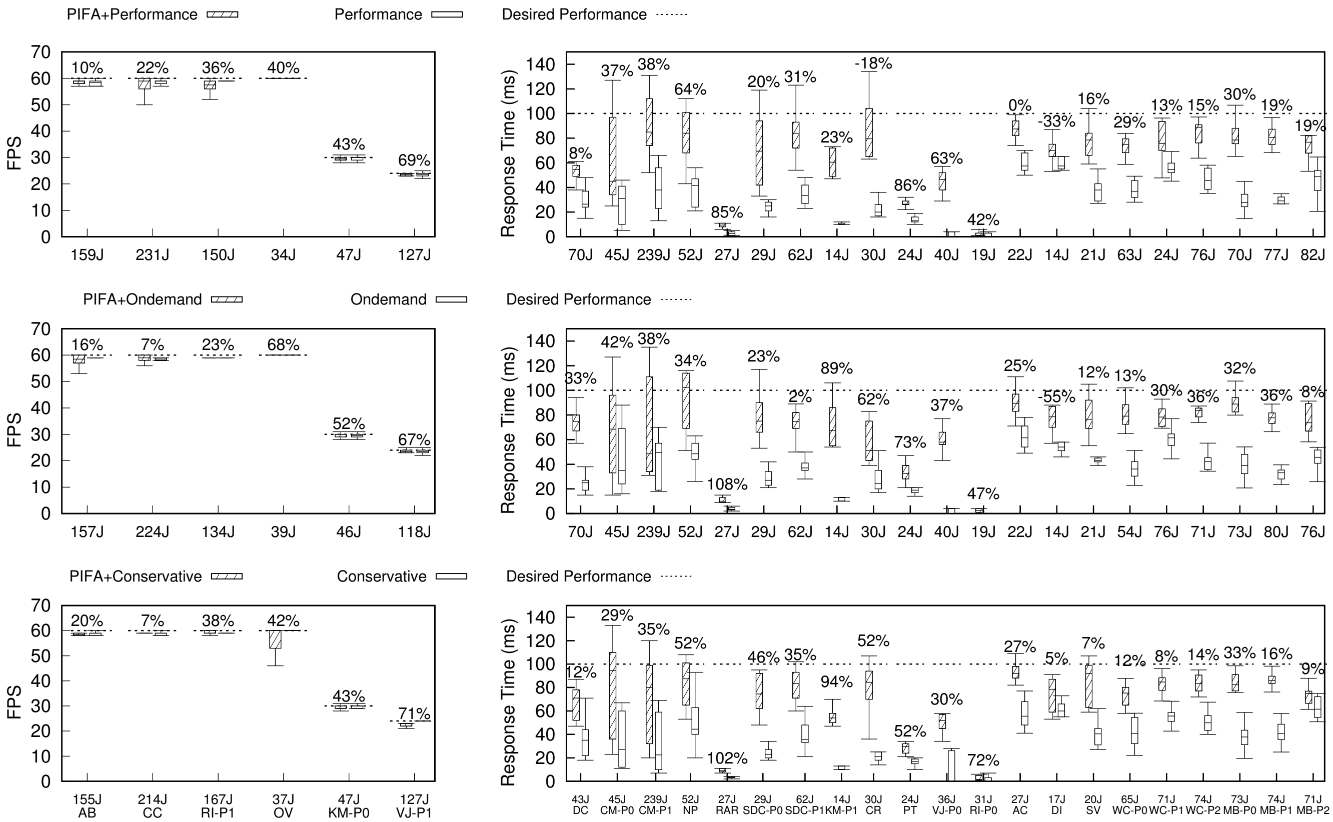


Fig. 4: The Apps from left to right: Angry Birds, Candy Crush, Replica Island(Phase1), OpenGLVideo, KM Player (Phase 0), Video John (Phase 1), Desk Clock, CM Security (Phase 0), CM Security (Phase 1), Note Pad, RAR, SD Card Cleaner (Phase 0), SD Card Cleaner (Phase 1), KM Player (Phase 1), Cool Reader , Power Tutor, Video John (Phase 0), Replica Island(Phase0), Alogcat, Did I, SubScale View, WeChat(Phase0-2), Mobike(Phase0-2). In the first (respectively, second and third) rows of graphs, the performance governor (ondemand governor and conservative governor) with and without applying PIFA are assumed. In the first (second) column of graphs, applications with the FPS (response time) performance metric are assumed.

more effective for game or video apps that are more resource-demanding and incur more workload, while being less effective for apps that incur rather low workload (e.g., UI apps). Also note that PIFA is able to intelligently choose not to apply frequency adjustment to the phases that incur low workload.

C. Overhead Incurred by PIFA

Table II shows the overhead incurred under PIFA for running each app. This overhead is defined to be the difference between the total energy used for running an app under PIFA and the total energy used for running the same app under exactly the same setting but without PIFA. As seen in Table II, 14 apps incur a reasonably low overhead (less than 5%), and 4 apps (“Angry Birds”, “Candy Crush”, “KM Player” and “Mobike”) incur higher overhead (around 6%). In particular, for the complex app WeChat, the energy overhead is just 4.12%, indicating PIFA’s practicality for current apps.

Summary for RQ2: In general, the overhead brought by PIFA is negligible, demonstrating the efficiency of the online analysis based on the model files produced by the offline analysis. Such results also demonstrate the advantage of our synergy approach that integrates the offline phase learning and discovery analysis and online phase identification and frequency adjustment analysis.

D. Evaluation of Phase Management

To evaluate the efficacy of the phase management, we compare two methods in this set of experiments, PIFA-performance and PIFA-performance without phase management, denoted by PIFA and “PIFA w/o Phase MGT”. As PIFA w/o Phase MGT does not differentiate multiple phases, for all the phases, we set an identical sweet frequency corresponding to the phase whose performance metric is FPS (for those apps whose performance metrics are both FPS and response time), or to the first identified phase (for those apps whose performance metric is only response time). We evaluate these two methods for the 7 multi-phases apps. In this set

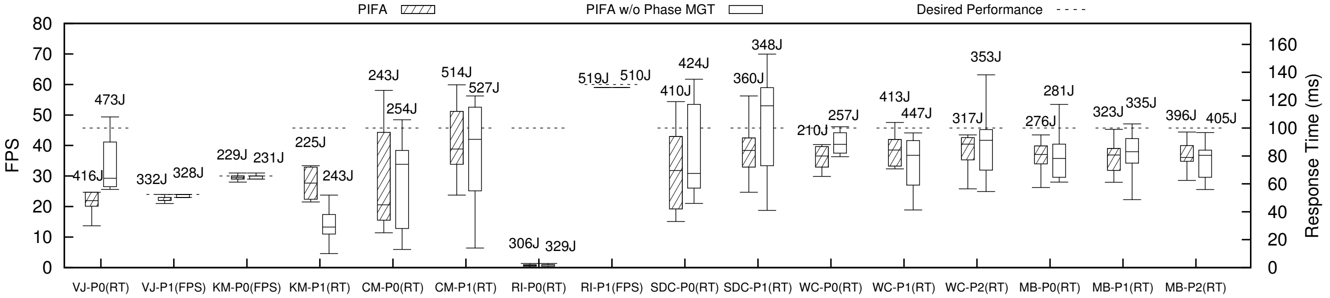


Fig. 5: Evaluation results of the phase control component. The x axis shows the specific phase of the multi-phase app and its associated performance metric. For each phase of an app, two boxplots show the results under PIFA and PIFA w/o Phase MGT, with the absolute consumed energy shown above the boxplot.

TABLE II: Overhead incurred under PIFA, shown in the format of energy overhead under PIFA / total energy under PIFA (ratio of the overhead over total energy)

App Name	ScaleView	Alogcat	Did I	OpenGL Video
Overhead (J)	≈ 0/74.20	≈ 0/111.70	≈ 0/101.88	12.87/367.95 (3.50%)
App Name	Angry Birds	Candy Crush	KM Player	ReplicaIsland
Overhead (J)	23.73/426.82 (5.56%)	31.44/523.78 (6.00%)	29.90/454.08 (6.58%)	30.52/686.73 (4.45%)
App Name	VideoJohn	Note Pad	RAR	Cool Reader
Overhead	9.24/745.18 (1.24%)	15.06/344.53 (4.53%)	11.64/316.23 (3.68%)	14.48/350.80 (4.13%)
App Name	Desk Clock	SD Card Cleaner	Power Tutor	CM Security
Overhead (J)	11.55/323.64 (3.57%)	27.68/763.83 (3.62%)	7.32/287.65 (2.55%)	19.85/616.40 (3.22%)
App Name	WeChat	Mobike		
Overhead (J)	11.35/275.38 (4.12%)	24.95/464.57 (5.37%)		

of experiments, the energy consumption is the device-level energy consumption measured when running the app under each method. Results are shown in Fig. 5 (the organization of which is explained in Fig. 5's caption). Note that since each tested phase either has an FPS metric or a response time metric (e.g., KM-P0 has a FPS metric and KM-P1 has a response time metric), we show both FPS (left y-axis) and response time performance (right y-axis) in Fig. 5.

As seen in Fig. 5, the phase management component is quite effective w.r.t. both performance and energy consumption. With this component, PIFA is able to accurately identify the execution phases, and set the frequency at runtime according to the sweet frequency learned offline for the corresponding phase. Additionally, PIFA often yields a performance close to or better than the desired performance while incurring close or less energy consumption than PIFA w/o Phase MGT which often yields a performance worse than the desired performance. For example, for the phase 0 of SD Card Cleaner app, PIFA yields a less energy consumption in phase 0 (410J) while achieving a performance better than the desired performance; whereas PIFA w/o Phase MGT yields a performance often worse than the desired performance. For the phase 1 of SD Card Cleaner app, PIFA actually yields a slightly higher energy consumption (12J more). This is because in this case, PIFA clearly yields a performance much better than PIFA w/o Phase MGT, whose performance is 16.1% deviated from the desired performance. This result implies that for certain phases, PIFA sets a more accurate sweet frequency that is higher than the frequency set by PIFA w/o Phase MGT, and thus consumes more energy. On the other hand, PIFA w/o Phase MGT simply sets a single frequency for all phases, saving more energy but causing performance violations.

Summary for RQ3: In general, PIFA is able to achieve better performance and energy efficiency than PIFA w/o Phase MGT. In some cases, PIFA w/o Phase MGT may cause performance violations due to lack of knowledge of the phases, but PIFA could guarantee stable performance with slightly more consumed energy.

E. Generalization to Other Device Models

We choose Nexus 6 (2.7GHz CPU and 3G memory) and Samsung S4 (1.6GHz CPU and 2G memory) as two alternative phone models to evaluate the generalization of PIFA. We use WeChat and Mobike for the experiment as both of them have three phases. First, we collect the test cases generated by DroidWalker by exploring the apps on Nexus 5. These test cases are used in the experiments for RQ1. We then run these test cases on Nexus 6 and Samsung S4, perform the offline clustering, and derive the sweet frequency for these two phone models. The results show that WeChat and Mobike are still clustered into three phases on both of the phone models and the sweet frequency for each phase is the same with Nexus 5. Finally, we evaluate the energy saving brought by PIFA by comparing PIFA with the on-demand governor, which is less energy intensive. Table III shows the results. We can see that PIFA saves energy on all the three phone models, but

TABLE III: Energy saving on different phone models

Model	WeChat					MoBike				
	Phase 0 (J)	Phase 1 (J)	Phase 2 (J)	Total (J)	Average (J)	Phase 0 (J)	Phase 1 (J)	Phase 2 (J)	Total (J)	Average (J)
Nexus 5	12 (13%)	39 (30%)	43 (35%)	94	31.33	39 (33%)	49 (36%)	10 (8%)	98	32.67
Nexus 6	25 (16%)	163 (57%)	35 (16%)	223	74.33	13 (14%)	62 (38%)	41 (22%)	116	38.67
Samsung S4	2 (12%)	11 (21%)	1 (5%)	14	4.67	7 (34%)	10 (39%)	4 (19%)	21	7

the amount of saved energy is different due to the different device configurations, such as screen size, CPU, and memory. Compared to the saving on Nexus 5, for WeChat, PIFA saves more energy on Nexus 6 in Phase 1 and less energy on S4 in Phase 3; for Mobike, PIFA saves more on S4 in all phases.

Summary for RQ4: PIFA can be easily generalized for different phone models. Since DroidWalker can replay the test cases across phone models, the test cases generated by DroidWalker on one phone model can be applied on other phone models. Based on the execution results of these test cases, PIFA can automatically infer the phases and sweet frequencies, saving energy for the apps on these phone models.

F. Evaluation Summary and Discussions

Our evaluations using real-world Android apps demonstrate that PIFA dramatically reduces energy consumption (e.g., $>$ 30% energy reduction for most apps) and incurs rather small runtime overhead (e.g., $<$ 5% overhead for most apps). In addition, PIFA achieves closely better performance than the desired performance specified for each phase. We should mention that PIFA depends on DroidWalker to explore app behaviors. Although the activity coverage of DroidWalker is much larger than the state-of-the-art Monkey for the evaluated apps (16% in the median case), there are still some behaviors of the apps that cannot be exposed by DroidWalker, e.g., those triggered by system events. As a result, the identified phases may not cover certain functionality of the app. To mitigate this issue, developers can provide their own UI test cases, written in Android user interfaces tests [1] or Robotium [4]. Combining the generated test cases of DroidWalker and the developer-written test cases can produce a more comprehensive test suite to cover more behaviors of apps, and thus make the models learned by the offline analysis achieve higher precision in identifying energy phases of the apps. Furthermore, PIFA focuses on optimizing CPU energy, since CPU is one of major source of energy consumption [35]. Considering other components simultaneously may yield further benefits, yet at the cost of more complicated design/implementation and runtime overheads, which may negate the benefits. In the experiments, we measured energy consumption for the entire machine, which shows our CPU-focused energy optimization techniques yield significant energy saving.

VI. Related Work

DVFS schemes. DVFS is an important power management technique that has been widely studied for improving energy efficiency in mobile devices. Existing DVFS algorithms mainly focus on the trade-off between the energy and real-time properties [33], [21], [26], [22], [31], [16], [11], [14], [34], [28], [39], [38]. For example, Pouwelse. et al. and Hamers et al. proposed several mechanisms to provide just enough computing capacity for decoding video frames [31], [16]. In [11], Chang et al. proposed a DVFS scheme to scale the CPU frequency based on a pre-defined resource usage model. Rao et al. presented an application-specific energy saving scheme for Android devices, which is yet phase-oblivious [34]. Generally speaking, Compared to these existing DVFS-based energy saving techniques, our key novelty is the combined offline-online analysis, allowing high accuracy in phase identification and low runtime overhead on DVFS adjustment. Prior works mostly focus on separate consideration of either offline clustering or online DVFS adjustment. Moreover, PIFA achieves full automation, an important contribution that scales PIFA to large numbers of apps and devices. To the best of our knowledge, no prior works on either DVFS or clustering provides such a synergistic, automated, and scalable solution. Therefore, we chose to compare against native Android power governors as seen in the experiments. We compare with PIFA w/o phase control, which actually yields similar performance as existing online DVFS techniques ([11], [34], [10], [19] since they share the same intuition.

Energy-efficient computing for mobile devices. A set of energy-efficient techniques have been proposed to explore various features of mobile computing, such as transmission delay characteristics [32], screen bright levels [7]. In [8], [23], several automatic techniques were proposed to detect behaviors that consume abnormally high system energy. A mobility prediction-based algorithm for smartphone was introduced to improve energy efficiency of daily location monitoring [12]. CPU-GPU cooperative power management techniques have been proposed to target at the case when both CPU and GPU are heavily used, such as 3D games [30], [18], [29]. Different from these works, we are minimizing system-wide energy consumption for running a wide range of apps. We focus on making the CPU frequency adjustment fully automated and intelligent, and combine offline analysis and online frequency scheduling to develop an app-stateful mobile power management framework.

VII. Conclusion

PIFA, an intelligent phase identification and frequency adjustment framework for energy-efficient and time-sensitive mobile computing, is presented. PIFA aims at improving energy efficiency of mobile computing while satisfying desirable latency performance required by various phases of an app. PIFA addresses two major challenges of accurately identifying different execution phases of an application and efficiently integrating the phase identification results for runtime frequency adjustment. Extensive experiments prove that PIFA dramatically reduces energy consumption while satisfying desired performance requirements with small runtime overhead.

References

[1] Automating user interface tests for android, 2017. https://developer.android.com/training/testing/ui-testing/index.html.

[2] F-droid app market, 2017. https://f-droid.org/.

[3] Google Play Store, 2017. https://play.google.com/store.

[4] Robotium: User scenario testing for android, 2017. http://www.robotium.org.

[5] Ui/application exerciser monkey, 2017. http://developer.android.com/tools/help/monkey.html.

[6] Trepn power profiler, 2019. https://developer.qualcomm.com/software/trepn-power-profiler.

[7] Anand, B., Thirugnanam, K., Sebastian, J., Kannan, P. G., Ananda, A. L., Chan, M. C., and Balan, R. K. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, pp. 57–70.

[8] Banerjee, A., Chong, L., Chattopadhyay, S., and Roychoudhury, A. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), pp. 588–598.

[9] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (2012), SOAP '12, ACM, pp. 27–38.

[10] Bezerra, P. T., Araujo, L. A., Ribeiro, G. B., Neto, A. C. d. S. B., Silva-Filho, A. G., Siebra, C. A., QB da Silva, F., Santos, A. L., Mascaro, A., and Costa, P. H. Dynamic frequency scaling on android platforms for energy consumption reduction. In *Proceedings of the 8th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks* (2013), ACM, pp. 189–196.

[11] Chang, Y.-M., Hsiu, P.-C., Chang, Y.-H., and Chang, C.-W. A resource-driven dvfs scheme for smart handheld devices. *ACM Trans. Embed. Comput. Syst. 13*, 3 (Dec. 2013), 53:1–53:22.

[12] Chon, Y., Talipov, E., Shin, H., and Cha, H. Mobility prediction-based smartphone energy optimization for everyday location monitoring. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems* (2011), SenSys '11, pp. 82–95.

[13] Choudhary, S. R., Gorla, A., and Orso, A. Automated test input generation for android: Are we there yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015* (2015), pp. 429–440.

[14] Gu, Y., and Chakraborty, S. A hybrid dvs scheme for interactive 3d games. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2008), pp. 3–12.

[15] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The weka data mining software: An update. *SIGKDD SIGKDD Explorations Newsletter 11*, 1 (2009), 10–18.

[16] Hamers, J., and Eeckhout, L. Exploiting media stream similarity for energy-efficient decoding and resource prediction. *ACM Trans. Embed. Comput. Syst. 11*, 1 (Apr. 2012), 2:1–2:25.

[17] Han, J., Kamber, M., and Pei, J. *Data Mining: Concepts and Techniques*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[18] Hsieh, C.-Y., Park, J.-G., Dutt, N., and Lim, S.-S. Memory-aware cooperative cpu-gpu dvfs governor for mobile games. In *Embedded Systems For Real-time Multimedia (ESTIMedia), 2015 13th IEEE Symposium on* (2015), IEEE, pp. 1–8.

[19] Jang, H. B., Kim, J. M., Lee, H.-J., Chung, S. W., Shin, Y., and Son, J. C. Intelligent governor for low-power mobile application processors. In *SoC Design Conference (ISOCC), 2013 International* (2013), IEEE, pp. 206–207.

[20] Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., and Wu, A. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 7 (2002), 881–892.

[21] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), MICRO 36.

[22] Liu, C., Li, J., Huang, W., Rubio, J., Speight, E., and Lin, X. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, ACM, pp. 23–32.

[23] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 57–70.

[24] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. Edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI* (2013), vol. 13, pp. 57–70.

[25] Ma, Y., Huang, Y., Hu, Z., Xiao, X., and Liu, X. Paladin: Automated generation of reproducible test cases for android apps. In *The 20th International Workshop on Mobile Computing Systems and Applications* (2019), HotMobile 2019.

[26] Mejia-Alvarez, P., Levner, E., and Mossé, D. Adaptive scheduling server for power-aware real-time tasks. *Journal ACM Transactions on Embedded Computing Systems (TECS) 3*, 2 (2004), 284–306.

[27] Miller, R. B. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (1968), AFIPS '68 (Fall, part I), pp. 267–277.

[28] Mishra, N., Zhang, H., Lafferty, J. D., and Hoffmann, H. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15, pp. 267–281.

[29] Pathania, A., Irimiea, A. E., Prakash, A., and Mitra, T. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *Proceedings of the 52Nd Annual Design Automation Conference* (2015), DAC '15, pp. 201:1–201:6.

[30] Pathania, A., Jiao, Q., Prakash, A., and Mitra, T. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference* (2014), pp. 40:1–40:6.

[31] Pouwelse, J., Langendoen, K., Lagendijk, I., and Sips, H. Power-aware video decoding. In *Proc. 22nd Picture Coding Symposium* (2001), pp. 303–306.

[32] Ra, M.-R., Paek, J., Sharma, A. B., Govindan, R., Krieger, M. H., and Neely, M. J. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, pp. 255–270.

[33] Rangan, K. K., Wei, G.-Y., and Brooks, D. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09, pp. 302–313.

[34] Rao, K., Wang, J., Yalamanchili, S., Wardi, Y., and Handong, Y. Application-specific performance-aware energy optimization on android mobile devices. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Feb. 2017), vol. 00, pp. 169–180.

[35] Shye, A., Scholbrock, B., and 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* (2009), MICRO 42, ACM, pp. 168–178.

[36] Vallee-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing java bytecode using the soot framework: Is it feasible? In *Proc. Compiler Construction* (2000).

[37] Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., and Xie, T. Automated test input generation for android: are we really there yet in an industrial case? In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016* (2016), pp. 987–992.

[38] Zhu, Y., Halpern, M., and Reddi, V. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *Proc. High Performance Computer Architecture (HPCA)* (2015), pp. 137–149.

[39] Zhu, Y., and Reddi, V. High-performance and energy-efficient mobile web browsing on big little systems. In *Proc. High Performance Computer Architecture (HPCA)* (2013), pp. 13–24.