Smartphone Analysis and Optimization based on User Activity Recognition

Yeseong Kim, Francesco Parterna, Sameer Tilak and Tajana S. Rosing
University of California, San Diego
email:{yek048, fpaterna, sameer, tajana}@ucsd.edu

***Abstract*—Behavior of smartphone systems is highly influenced by user interactions, such as 'zooming' and 'scrolling', which determine the execution phases within applications and lead to different power and performance demands. Current power and thermal management algorithms are agnostic to these behaviors. We propose a novel user activity recognition framework that enables user activity-aware system decisions. The proposed framework carefully monitors system events initiated by user interactions and identifies the current user activity based on an online activity model. We implemented the proposed framework in Android platform, and tested it on Qualcomm MDP 8660 smartphone. To show the practical value of our recognition strategy, we design effective power and thermal management policies that adapt system settings to user activity changes. Our experimental results using 10 real mobile applications show that the proposed proactive management technique can reduce the CPU energy by up to 28% while meeting a given thermal constraint.**

I. Introduction

Today's smartphones run many different applications, or apps. Since user's activities initiate different capabilities in an app, the app's system resource usage varies significantly. As an example, Figure 1 shows our measurements of Gallery app, a default Google image viewer. Three different user activities are highlighted: app launch, scrolling images, and zooming an image. When scrolling images, the amount of storage access is relatively small since the thumbnails of images are already loaded from flash memory during the app launch. On the other hand, for the zoom in/out, the CPU utilization and flash memory bandwidth are very high since the app resizes images while loading the details of the image.

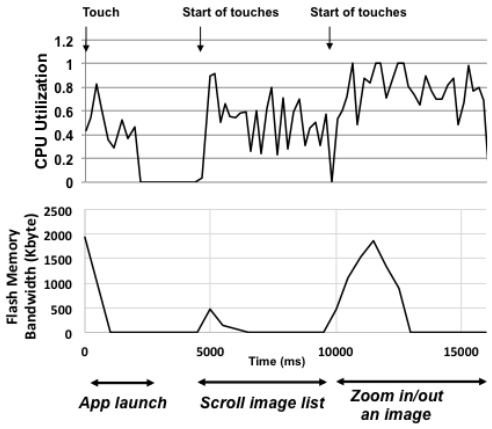

Fig. 1. Smartphone CPU/flash memory usage changes initiated by user interactions (Gallery app)

Smartphones pass through diverse user activity phases which are initiated by user interactions. This insight has not been used to date for power and thermal management in mobile devices, even though leveraging such information may lead to more efficient power and thermal management decisions. For example, SmartCap [1] proposed a CPU power management technique which identifies different minimal required frequencies for smartphone apps. However, as shown in Figure 1, CPU demands for various user activities are quite different during a single app's execution, so being able to adapt intelligently while executing a particular app would give more effective results.

In this paper, we propose a novel automated user activity recognition framework and implement it in Android 4.0.1. By monitoring how each user interaction initiates its related tasks, the proposed framework automatically builds a user activity model for all apps. Based on the model, the framework provides a way to recognize each of user's activities at runtime with negligible overhead. In order to illustrate the effectiveness of our technique, we also design a user activity-aware power and thermal management policies to proactively adapt the CPU frequency. The experimental results conducted on a Qualcomm MDP8660 smartphone [17] with ten Android applications show that the proposed user activity-aware power management technique can save CPU energy by up to 28% and 9.5% on average over the unmodified default Linux CPU governor and SmartCap, respectively, while our thermal management policy can proactively ensure given thermal constraints are met.

II. Related Work

User's behavior is highly correlated to system activities of smartphones. Thus, many research groups have focused on understanding user's behavior and exploiting the behavioral characteristics to improve smartphone systems. For example, FALCON [2] suggested an app launching prediction technique by learning when and where users have used their apps. Based on their prediction model, they proposed a prefetching technique which can reduce app launching times. Falaki et al. [3] characterized how smartphone users interact with their smartphone over different locations and time, and analyzed the resulting impact on networking subsystem and battery. They also demonstrated that the users' behavior patterns can be usefully exploited for predicting the future energy usage. Their observations agreed with the findings of Shye et al. [4]. However, our work is quite different from the previous work as we focus directly on how apps react to activities which initiates diverse user interactions for smartphone apps.

A number of research groups have investigated how to systematically monitor apps due to user interactions. For

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

example, Qian et al. [5] presented a network usage analysis tool, called ARO. ARO exposes cross-layer interactions from the user interaction layer to the hardware radio resource management. ProfileDroid [6] proposed a profiler which assesses apps at different layers such as user, OS, and hardware layers. While both of these approaches are helpful in analyzing user interactions for a specific purpose such as network usage, our strategy is more general in that our phase recognition technique is the basis of every user's interaction with the smartphones, and thus can be used for general-purpose system dynamic management, not limited to analysis.

III. Activity Phase Recognition

In this section we describe our Android-based automated user activity phase recognition framework. There are two key challenges in supporting the user activity recognition functionality in a system software without any prior knowledge of app behaviors. First, the framework should selectively profile system-level information which can represent a semantic meaning of a user activity such as taking a picture by pressing a button. Second, in analyzing the profiled user activity information, the framework should demonstrate a relationship of user activities with the right abstraction model to effectively distinguish similar activities from the current one. To address these challenges, our technique first monitors which functions of an app are invoked from a user interaction, to capture the meaning of the user's activity. By comparing the monitored function calls for each user interaction, the framework builds a *user activity model* that shows how user activities may be correlated. Then, it recognizes the current user's activity by finding the most similar ones in the model.

Figure 2 shows the architecture of our proposed technique. The main module is the Activity Phase Recognition engine (APR engine) which is responsible for building the user activity model and identifying the current activity. The APR engine consists of three submodules, user activity profiler (UAProfiler, Section III.A), user activity model builder (UABuiler, Section III.B), and user activity recognizer (UARecognizer, Section III.C). Whenever a user generates an interaction event such as touching a button on the screen or pressing a back button of the hardware keys, the function call tracer module begins tracking function calls which are initiated from this interaction. The tracked functions are sent to the UAProfiler module where the collected functions for user's interaction are composed into the current *activity profile*. As soon as the UAProfiler module finishes collecting the current activity profile, the UABuiler module updates a *user activity model* of each running app. The model represents the relationship of different user activities as a clustered structure of the extracted profiles. Based on the user activity model, the UARecognizer module tries to identify the current user's activity. The UAProfiler module also provides a partially-completed current activity profile, called an *intermediate activity profile*. Given the user activity model and the intermediate activity profile, the UARecognizer module identifies which user activity from the learned model is most similar to the current profile. The identified user's activity can be sent to other modules such as activity phase-aware analysis engine and activity phase-aware system management engine as discussed in Sections IV and V. We next discuss the details of each of APR engines' components.

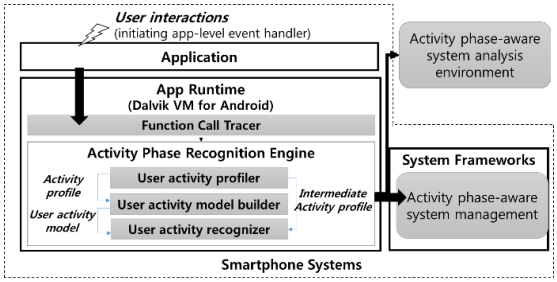

Fig. 2. An architectural overview of the proposed activity phase recognition system

A. User Activity Profiler

The UAProfiler module extracts an *activity profile* for each user's interaction. We exploit that fact that a meaningful user's activity can be represented by the functions invoked from the user's interaction. For example, if a user touches a screen to focus the camera, the touch interaction initiates consecutive function calls which are responsible for processing the focusing activity. In mobile systems, user interactions are typically processed by event handlers [7, 20], which are special functions that handle different types of user interactions such as touch and key press events. Thus, the UAProfiler module extracts the activity profile in the form of the invoked functions which are initiated from an event handler.

In order to make an activity profile for each user's interaction, the UAProfiler module first collects function calls invoked from an event handler, and generates a single representative form that summarizes which functions are related for each user's activity. More formally, once the i-th user interaction initiates an event handler $\mathbf{e_i}$, $\mathbf{e_i}$ invokes function calls $\mathbf{F_i} = (\mathbf{f_1^i}, \mathbf{f_2^i}, \ldots \mathbf{f_N^i})$ where each $\mathbf{f_X^i}$ represents functions implemented by the app developer. $\mathbf{F_i}$ includes all function calls which are directly invoked from $\mathbf{e_i}$ in a single thread $\mathbf{t_1^i}$ until the event handler returns. Additionally, the event handler $\mathbf{e_i}$ can initiate other function calls over different thread executions, say $\mathbf{t_2^i}, \mathbf{t_3^i}, \ldots, \mathbf{t_M^i}$, including either new thread invocations or new event handlers generated by message sending/handling mechanism [8]. Thus, this module also includes the function calls of all the initiated threads into $\mathbf{F_i}$. For an event handler $\mathbf{e_i}$, we call the running period between the start of the event handler and the end of all associated threads $\mathbf{t_j^i}$ (where $1 \leq \mathbf{j} \leq \mathbf{M}$) an *activity phase* p_i. Since some threads could be locked while running a loop, we also assume that a thread $\mathbf{t_j^i}$ is finished if the thread is locked.

For an activity phase p_i, the UAProfiler module elaborates its *activity profile* V_{p_i} using the invoked function calls $\mathbf{F_i}$. Although an activity phase could include a large number of function calls, we have observed that the meaning of a user's activity can be identified by using only a smaller number of semantically important function calls at the beginning of the activity phase. For example, focusing camera view initiates newAutoFocusCall() and onAutoFocus() as the first and second function calls. Thus, in order to reduce the computation complexity for user activity identification, we collect the $\mathbf{N_{func}}$ earliest function calls without consideration of the function order. That is, for a given function call set $\mathbf{F_i} = (\mathbf{f_1^i}, \mathbf{f_2^i}, \ldots \mathbf{f_N^i})$ and $\mathbf{N} \leq \mathbf{N}_{func}$, an activity profile for p_i is

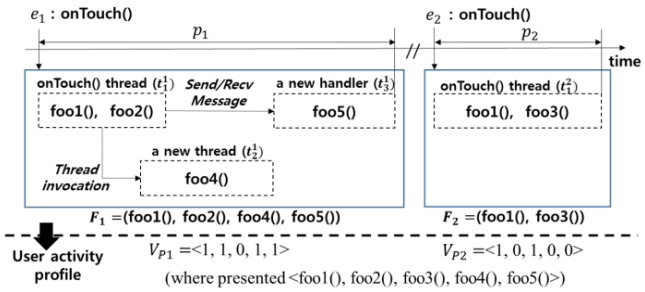

Fig. 3. An example of user activity profile extraction

represented as a binary vector $V_{p_i} =< v_1, v_2, \ldots, v_k, \ldots, v_L >$ where L is the total number of all observed app functions formed to each vector element and an element v_k represents its occurrence in F_i as either 0 or 1. Even though we don't consider the order of functions, this is sufficient to distinguish different user activities because different meaning of user activities exhibit very distinct function sets.

Figure 3 shows an example of the functions that are collected in building the user activity profile. There are two activity phases p_1 and p_2. Once user touches the screen for the activity phase p_1, an event handler e_1 named "onTouch()" invokes subsequent functions, foo1() and foo2(), in t_1^1. Because the event handler also invokes a new thread t_2^1 and a new message handler t_3^1, we track the function calls of the thread executions (i.e. *foo4* and *foo5*). It also tracks the function calls for p_2 in the same way. After extracting the two activity profiles, the represented binary vectors for p_1 and p_2 are V_{p1} =<1, 1, 0, 1, 1> & V_{p2} =<1, 0, 1, 0, 0> where the elements of a binary vector indicate the invocation of functions in the set, i.e., *<foo1, foo2, foo3, foo4, foo5>*.

B. User Activity Model Builder

The UABuilder module classifies all activity profiles by building an abstracted *user activity model* for each app. Before describing this module in detail, we first demonstrate the conceptual abstraction model for user activities: what user activities are ***identical*** in the sense of the semantic meaning? For example, when a smartphone user takes a picture twice by touching a button on a Camera app screen, we can observe two user activities, A1 and A2. If there is no significant difference in their semantic meaning, we can say that the two user activities are identical. However, if A1 stores the picture in the flash memory in the JPEG format while A2 stores its picture in the PNG format, these activities would be considered identical by some and different by others. Thus, there is no simple unbiased way to decide if they are identical or not.

Due to such subjective view in defining the identical activity, we introduce a hierarchical clustering-based model rather than strictly defining semantically identical user activities. For example, if A1 and A2 are represented by two slightly-different activity profiles, they could be clustered in a single cluster, meaning that they are strongly correlated. The proposed clustering-based user activity model can serve diverse system analysis and optimizations. For example, a user activity-aware CPU management policy can choose a proper clustering level for controlling the CPU frequency to achieve high energy efficiency by considering the resource usage similarity for different user activities. We demonstrate the benefit of the clustering-based model in Section IV.

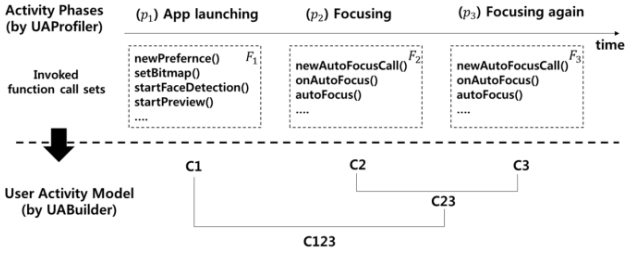

Fig. 4. An example of user activity model building (Camera App)

The activity model is motivated by our observation that similar user activities involve a similar set of function calls. Figure 4 illustrates the procedure of building a user activity model for Camera app. In this example, user interactions generate three different activity phases, p_1, p_2 and p_3, which are associated to three activity profiles. Since the activity phases for p_2 and p_3 are executed for the focusing user activity, we observed that the function call sets, F_2 and F_3, are very similar. On the other hand, the user activity phase p_1 is very different from either p_2 or p_3, as illustrated in their function call sets. We build the user activity model by considering how similar the two different activities are.

In order to compare similarity between two activity profiles represented as two binary vectors, we use the Jaccard Index [10]. The Jaccard Index is binary vector comparison criteria that computes the similarity as follows:

$$J\left(V_{p_i}, V_{p_j}\right) = \frac{|V_{p_i} \cap V_{p_j}|}{|V_{p_i} \cup V_{p_j}|}.$$

Intuitively, if p_i executes very similar activity as p_j, $J\left(V_{p_i}, V_{p_j}\right)$ has a large value. For example, for two activity profiles $V_{p_1} =< 1,1,1,0,1,0,0,0 >$ and $V_{p_2} =< 1,1,1,1,1,0,0,0 >$, the Jaccard index is 4/5, meaning they are similar activities. If the activity profile V_{p_i} is the same as V_{p_j}, Jaccard index is 1.

Using the Jaccard index similarity criteria, the UABuilder module builds our hierarchical structural model from activity profiles using hierarchical *complete-linkage clustering algorithm* [9]. We choose this algorithm among the hierarchical clustering algorithm family since it is one of the simplest and fastest. For example, compared to other hierarchical clustering algorithms such as Ward clustering, the complete-linkage clustering algorithm can be executed with a smaller number of comparisons. Starting from an initial clustering where each activity profile is considered as a separated cluster, this clustering algorithm progressively combines the two most similar clusters into a new cluster. If the compared cluster includes more than two activity profiles, this algorithm computes similarities between every pair of the two profiles for each cluster and considers the minimum similarity as the similarity of two clusters. This step is repeated until all activity profiles are combined into a largest single cluster. In the example shown in Figure 4, ***C2*** and ***C3*** are first combined into a single cluster, ***C23***, because ***C2*** and ***C3*** represent the most similar activity profiles. Then the

algorithm constructs the largest cluster ***C123*** by combining ***C23*** and ***C1***. Consequently, the final clustering result is the hierarchical user activity model.

In order to identify which subclusters similarly behave to the system components such as CPU/GPU utilization and memory bandwidth, we exploit *t-test* methodology [21], which is a well known statistic used to determine whether two sets of sampled data are significantly similar to each other. With a threshold parameter, called *confidence interval*, the t-test produces a *p-value* and evaluates the difference between data comparing the p-value with the *critical region*. The critical region is predefined by a confidence interval. For example, if we set the confidence interval threshold to 95%, the critical region is given by 1.960. If the p-value of the two sample data set is computed to be 4.0, since it is larger than the critical region (i.e., 4.0 > 1.96), we can assume that there is significant evidence that the two sets are statistically different.

Starting from every end node (cluster) of the hierarchical model, we repeatedly apply the t-test to the sampled system usage data of the two clusters. Assuming that a cluster $\boldsymbol{C}$ has two children, $\boldsymbol{C}_{left}$ and $\boldsymbol{C}_{right}$, if there is statistical difference between $\boldsymbol{C}_{left}$ and $\boldsymbol{C}_{right}$, we determine that the two clusters have two different activity phases of interest, called Significant Cluster or *SCluster*. Otherwise, we further test the parent cluster of $\boldsymbol{C}$ using the same methodology until getting to the root node. For example, in the previous example of Figure 4, if ***C2*** and ***C3*** are statistically similar while ***C1*** is different from them, we select ***C1*** and ***C23*** as the SClusters.

C. User Activity Recognizer

Using the constructed user activity model, the UARecognizer module identifies which activity phase in the past is the most similar to the current activity phases at runtime. In order to recognize the current user activity as soon as possible, the UAProfiler module also provides the current *intermediate activity profile* which is also represented as a binary vector. For example, while running an activity phase p_i, if three function calls, f_1^i, f_2^i and f_3^i, are monitored, an intermediate binary vector V'_{p_i} is constructed from the monitored function calls, and it is sent to UARecognizer module. Thus, recognitions can begin at the startup of each activity phase, before the activity profiler is completely matched in the constructed model.

Then, by exploiting the intermediate activity profile V'_{p_i}, the UARecognizer module can find the most similar activity phase of the learned user activity model to p_i. For a given user activity model built from the past activity profiles (V_{p1}, $\dots, V_{p_j}, \dots, V_{p_{i-1}}$), if V_{p_j} is the most similar one to V'_{p_i} based on the Jaccard Index, the hierarchical clusters which includes V_{p_j} is selected as the clusters of the current activity phase. Finally, the clustering information is sent to other external analysis frameworks and optimization modules. In an example shown in Figure 4, if P_3 is the most similar phase to the current running phase, this module sends the clustering information including the clusters, ***C3***, ***C23***, and ***C123*** with all the activity profiles of each cluster and its SCluster.

IV. Activity Phase-Aware Analysis

A. Implementation

We implemented the proposed framework as a module of Dalvik VM in Android 2.3 running on Qualcomm MDP8660 [17]. We instrumented code for handling function calls and returns in the original Dalvik VM implementation. We track user interactions via touch and key press event handlers, such as dispatchTouchEvent() and performClick(). Even if an activity phase is initiated by different users, there is no difference in the recognized activity profiles since the proposed framework recognizes activity using the sequence of functions in a systematical way. For example, the focusing activity of Camera app used by different users is represented as a single profile.

In order to select the maximum number of function calls, N_{func}, which the UAProfiler module collects, we investigate how many functions calls should be tracked to sufficiently recognize user's activity phases. We found that different user's activities can be represented by using up to 16 function calls. For example, an activity profile, which represents loading a webpage activity of the Android default browser app, can be recognized when the onPageStarted() function is initiated as the seventh function of its immediate activity profile. Thus, for all of experiments we set N_{func} value to 20.

The current implementation of our technique has been optimized so that users do not notice any overhead when our technique is enabled. Since Dalvik VM already keeps track of the function calls to maintain their runtime stack information, the time overhead for function call tracking is negligible. We store the model that the UABuilder module generated, and incrementally update the model if the current extracted activity profile does not appear in the past. Since the hierarchical model is updated only when a new binary vector is observed, the updates are not frequent. In our experiments, when the model is updated, different activity profiles are represented using about 100 binary vector elements (4 integers), and the total time overhead for updating a hierarchical model is less than 30ms.

The proposed technique can also be easily ported to any mobile devices. The platform-dependent part of the technique is only the function call tracer module. For example, for Windows mobile phone app support, it would be possible to extract function calls while interpreting MSIL byte code [11] in an instrumentation fashion.

B. Evaluation of Activity Model

To make a robust analysis of our activity phase recognition technique in distinguishing different user activities, we used ten different Android apps listed in Table I. Among the apps, five are the top popular ones on Google Play, four are taken from popular Android default apps, and the last app has been chosen to focus more on 3D interactions. Each app was executed for 5 minutes while our framework was active. We compare a manual classification of user behavior to the result obtained with the proposed automated procedure.

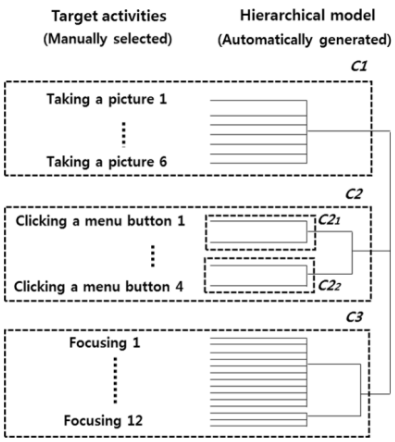

Fig. 5. An validation example of the automated activity phase recognition technique (Camera App)

Figure 5 describes the hierarchical model for Camera app along with the manually labeled user's activities. The result shows that the computed clusters match well the manual labels. For example, the cluster $\boldsymbol{C1}$ represents the taking a picture activity while the clusters $\boldsymbol{C2}$ and $\boldsymbol{C3}$ represent clicking a menu button and focusing activities. This model also shows how our technique can interpret the hierarchical semantic similarity. For example, clicking a menu button activity for opening and closing the menu are represented as clusters $\boldsymbol{C2_1}$ and $\boldsymbol{C2_2}$, respectively.

We further validate results of the hierarchical activity models of the other selected apps. Our proposed technique also successfully recognizes user's activities for the apps in a similar fashion. Using the t-test methodology with 95% of the confidential interval threshold, we could find on average 6 significant clusters (SClusters) per app which differ from each other in terms of CPU/GPU utilization and eMMC bandwidth. We describe the detailed characteristics of all identified SClusters in Section IV.D. We also select two representative user activity phases, which were most frequently observed in the collected log of each app. Table I lists the apps and the representative activities which were recognized in the hierarchical model. In the rest of the paper, we use the label of each activity phase as shown in Table I.

C. Activity-specific Resource Usage Analysis

Since the proposed framework can recognize different user activities in an automated way, we can analyze the system behavior on the basis of activity phases. In this section, we describe the characteristics of recognized user activities, and discuss the feasibility of activity-aware system management.

1) Energy consumption breakdown

In order to understand how user activities affect the application energy consumption, we performed an activity-specific energy analysis. We collect the total power consumption using Qualcomm **trepn** profiler, which provides accurate power consumption measurements from hardware energy counters, while executing the apps of Table I for 5 minutes. For each app, we track the energy consumed by the recognized activity phases represented as SClusters. Note that since the framework can recognize all activities in system software, we can analyze the fine-grained energy consumption of each activity phase without any manual effort.

TABLE I. 20 REPRESENTATIVE ACTIVITY PHASES

Test App	Representative recognized user activities
Facebook	Scrolling posts **(F1)** Opening an image **(F2)**
Messenger (Facebook)	Scrolling a contact list **(M1)** Sending an message **(M2)**
Pandora	Scrolling a station list **(P1)** Playing a music **(P2)**
Instagram	Scrolling image posts **(I1)** Loading new posts **(I2)**
Flash light	Turning the light on/off **(L1)** Changing the light type **(L2)**
Camera	Focusing the camera **(C1)** Taking a picture **(C2)**
Gallery	Scrolling an image list **(G1)** Zooming an image **(G2)**
Email	Scrolling an email list **(E1)** Checking new emails **(E2)**
Browser	Scrolling a webpage **(B1)** Loading a webpage **(B2)**
Neocore (3D Demo)	Playing 3D game demo **(N1)** Opening menu **(N2)**

Figure 6 shows the breakdown of energy consumption for the two representative activities of Table I, out of all recognized user activities represented as SClusters. The rest includes all other recognized activities and idle periods. The result shows that the mobile energy consumption is highly influenced by user interactions. A significant portion of energy is consumed by frequent user interactions. For example, the Facebook app consumes 50% energy consumption when scrolling (i.e., Activity F1). On the other hand, less interactive apps, which mostly use hardware components (e.g., Flash light and Camera app), consume a large portion of energy due to other components, such as camera.

2) Activity-specific system resource usage

In order to understand the impact of different usage phases on various system resources, we conducted a further quantitative activity phase-aware analysis. We measured the usage of CPU and GPU via a custom measurement tool based on sysfs, and of the bandwidth of eMMC flash storage via **iostat** tool. We classified the collected logs according to each recognized activity phase in the same fashion described above.

Figure 7 shows that, even for the same app, different user activities may exhibit very different characteristics for the HW components. For example, in the Pandora app scrolling a station list (P1) shows 1.63x higher CPU utilization compared to playing music (P2) due to many computations needed to update the screen. On the other hand, the Flash storage bandwidth for P2 is higher than P1 since P2 has to cache the downloaded music in the storage. We can observe these different characteristics over different activity phases.

In addition, although activity phases may behave quite differently, we have observed that most of activities exhibit very unique patterns whose system resource usage does not change much. To better understand this, we partitioned the collected log of each app into 2 sub logs. Since each collected log are 5 minute-long, we could create two 2.5 minute-long logs, say, the first half and the second half. Figure 8 shows the comparison between separated logs for CPU utilization of two different user activities, G1 and M2. The result shows that workload per user's activity changes little. For example, G1 activity requires about 50% of CPU utilization consistently over time while M2 mostly demands 80% of CPU utilizations.

D. System-wide Activity Characteristics

As described in Section IV.C, the recognized activities present two important aspects for system behavior. First, different activities may exhibit significant difference for the

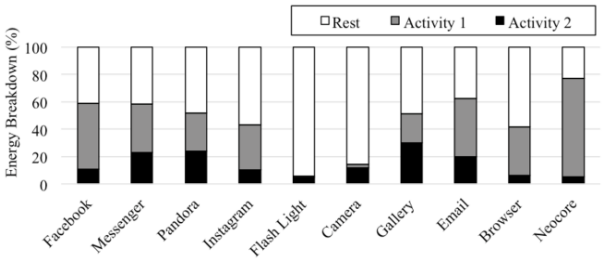

Fig. 6. Energy breakdown of 10 experimented apps

system resource usage. Second, there is a unique usage trend for each activity. In order to verify whether these aspects are common for every recognized activity, we have collected all SClusters and evaluated their system characteristics.

Our framework selects multiple SClusters for each app. While a typical app has six SClusters on average, Facebook app has ten SClsuters. Since SClusters are identified based on the statistical difference, it means that different activities represented as SClusters behave very differently even they belong to the same app, confirming the first aspect. In order to validate if the unique trend for each SClusters is also generally observed, we performed the same partitioning methodology described in Section IV.C.2, i.e., for a given single SCluster, we partitioned the collected utilizations and bandwidths into two sets. We applied the t-test to compute p-value of the two sets. Figure 9 summarizes the p values of the identified SClusters whose collected number of sample data is greater than 40, which is known as a large enough sample size [21]. The result shows that the unique characteristics of the activity phases exist in system-wide manner. For example, if we set the confidence interval to 95%, as denoted in the figure, the CPU/GPU utilizations exhibit statistically similar characteristics for more than 80% of activity phase.

Since our framework can automatically capture the unique trend of workload characteristics for each user activity represented as SClusters, we can leverage the recognized activity phases to design automated system management.

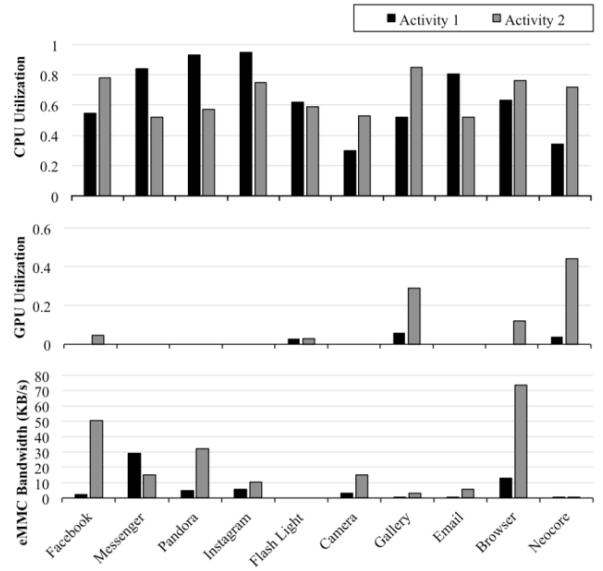

Fig. 7. An activity phase-specific system resource usage analysis

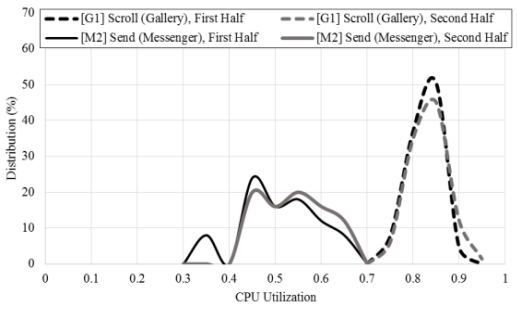

Fig. 8. Distribution of CPU utilization for two activity phases

V. Activity-aware Management

In this section, we describe power and thermal management techniques which leverage the user's activity recognition framework. The management technique that has been implemented in Android platform consists of two parts. The first one is a power management strategy, named PhaseCap, which sets the minimum CPU frequency of each phase according to the user experience requirements provided by SmartCap [1]. The PhaseCap's goal is to reduce power as much as possible without degrading the user's experience. Thanks to the phase recognition, PhaseCap has better adaptive properties to save energy battery with respect to the state-of-the art solution [1]. The second component is thermal management which works in tandem and adjusts the frequency set by PhaseCap when the temperature is predicted to be higher than a safe threshold. The architectural overview of our management strategy is illustrated in Figure 10.

Similar to the previous section, we implemented our power and thermal management techniques along with the activity recognition framework on the Qualcomm msm8660 Android smartphone [17]. These two techniques manage power and temperature of the device by controlling CPU frequency via sysfs interfaces, running Linux Kernel 3.0.8. We run ten apps of Table I on the phone, while measuring energy using Qualcomm **trepn** tool [18], and reading the temperature from the embedded sensors via sysfs interfaces. We use a user interaction replay tool [22] to ensure a fair comparison for different policies.

A. Power Management

The proposed PhaseCap technique leverages our activity recognition framework. As we discussed in Section IV.C, apps are characterized by a specific number of phases represented

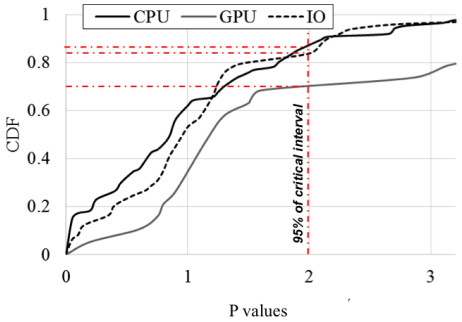

Fig. 9. Distribution of p values for identified SClusers

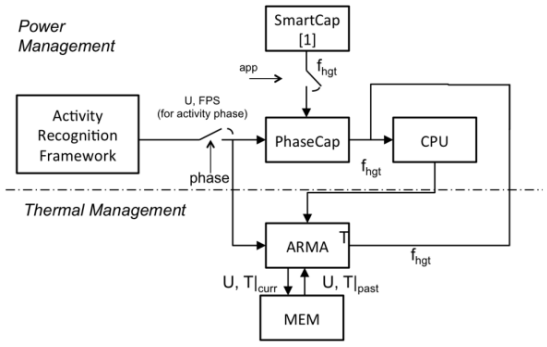

Fig. 10. Activity phase-aware power and thermal management

as SClusters which significantly differ with each other in terms of system resource usages. Thus, PhaseCap makes decisions for each user activity phase by adjusting the CPU frequency while considering the unique characteristics of the SClusters recognized by our activity recognition framework.

In contrast, the recently proposed SmartCap strategy sets the minimum frequency, which satisfies the user experience needs for the duration of the run of the whole app, f_{app}, to the highest frequency of the CPU governor. The set of frequencies $\{f_{app\text{-}1}, f_{app\text{-}2}, \ldots, f_{app\text{-}N}\}$ corresponding to a set of apps $\{app\text{-}_1, app\text{-}_2, \ldots, app\text{-}_N\}$ is computed based on user studies [1]. Then, the highest frequency of the default Linux CPU governor of the Android platform such as *ondemand* or *interactive* governor is restricted by the highest frequency f_{app-i}.

The proposed PhaseCap improves on SmartCap by instead setting the frequency as a function of the application phase. PhaseCap computes the minimum frequency f_{hgt} for the app $app\text{-}_i$, at phase *ph-j*, to $f_{ph\text{-}j}$ according to Equation (1) in which $U_{ph\text{-}j}$ is the CPU utilization at the phase *ph-j*.

$$f_{ph-j} = f_{app-i} \cdot \frac{U_{ph-j}}{\max_k(U_{ph-k})} \qquad (1)$$

We adjust the computed f_{ph-j} based on the fact that user experience can be estimated with Frame per Second (FPS) [19]. PhaseCap keeps monitoring the current FPS, and if the FPS is less than the maximum FPS, FPS_{ph-j}, observed by the framework, it incrementally increases the frequency f_{ph-j}. Due to the sub-phase frequency reduction operated by PhaseCap, power is saved while guaranteeing user satisfaction.

In our experiment, we used ten apps of Table I to evaluate the activity-aware power management. We first used each app for 5 minutes to learn online the activity model, and then replayed a user interaction log of the app for another 5 minutes while activating PhaseCap. The interaction logs included different interaction sequences from the first learning step.

Figure 11 presents a result of an example snippet for Gallery applications over three policies, the unmodified *ondemand* governor, *conservative* governor, PhaseCap. The result shows that by considering the three different characteristics of user activities, PhaseCap efficiently applies different CPU frequencies for different activities, while not violating the FPS compared to the default CPU governor. In addition, while the conservative governor often misses the user experience requirement at the beginning of the user interactions, PhaseCap adapts quickly to user activity changes.

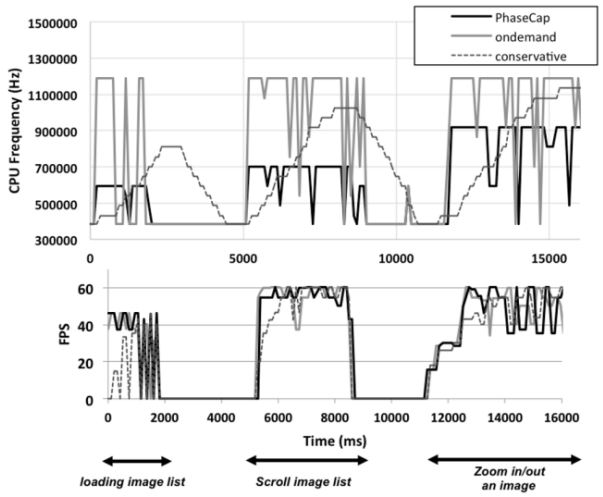

Fig. 11. An example of the activity phase-aware CPU frequency management policy (Gallery app)

Figure 12 presents energy saving and FPS comparison results of the PhaseCap using the unmodified default Linux *ondemand* governor as the baseline. The result shows that our PhaseCap achieves higher savings compared to the baseline. For example, average energy savings of Messenger app is 28%. We also compared the PhaseCap to the original SmartCap. The SmartCap governor sets the highest frequency of the ondemand governor to a single value for the whole execution time of an app. Our fine-grained PhaseCap governor always outperforms SmartCap. For example, PhaseCap save 9.5% more energy for Email app over SmartCap. In addition, there is no significant difference in FPS. Thus, we conclude that the activity-aware management mechanism is very important in achieving high energy efficiency.

B. Thermal Management

Saving power is important not only to extend the battery's lifetime but also to keep the system's temperature envelope below a threshold. High temperatures must be avoided for meeting reliability constraint [12] and for the human skin that can tolerate up to 45°C [13]. Since proactive thermal management has been shown to be very effective in servers, we leverage the Auto Regressive Moving Average (ARMA) model used in [14], but instead we apply the activity phases that our framework detects online in order to figure out the workload of the near future.

Let T_k be the temperature and U_k the CPU utilization at instant k. The temperature at the instant $k+1$ can be predicted using Equation (2) in which the coefficients (a_i, b_j, h, l) are estimated during a training stage. The temperatures $(T_k, \ldots, T_{k\text{-}l})$ and the utilization $(U_k, \ldots, U_{k\text{-}h})$ are read from sensors and performance counters while the future utilization U_{k+1} is provided by the recognition mechanism as the same way of the power management, i.e., $U_{ph\text{-}j}$ for the phase ph_j at that time.

$$T_{k+1} = T_k \cdot a_1 + T_{k-1} \cdot a_2 + \cdots T_{k-l} \cdot a_{l+1} + U_{k+1} \cdot b_0 + U_k \cdot b_1 + \cdots + U_{k-h} \cdot b_{h+1} \qquad (2)$$

The predicted temperature T_{k+1} is compared with the safe threshold T_{safe}. If $(T_{k+1} < T_{safe})$ no action is taken, otherwise the minimum frequency computed by the power management is

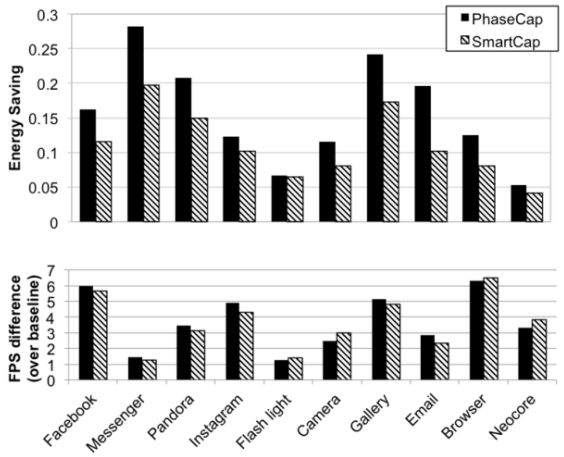

Fig. 12. Energy saving of the activity phase-aware CPU power management policy

updated to the new value $f_{min\text{-}p}$ according to Equation (3) in which the coefficient c is obtained by simple scaling as shown in Equation (4), since utilization and power are correlated [15] and the temperature is a function of the dissipated power.

$$f_{min-p} = f_{min} \cdot c \quad (3)$$

$$c = \frac{|T_{k-1} - T_{safe}|}{T_{safe}} \quad (4)$$

Every time the predicted temperature differs from the read one by 1°C or more, the ARMA model is updated. We found the *(l,h)* pair in an offline manner, and the (a_i, b_j) remaining parameters are computed at runtime using a least-square fit. In our evaluation, we monitor the utilization and the temperature for previous 5 seconds at 100ms resolution (i.e., l=h=50) since it showed the lowest error for different *(l,h)* configurations with negligible overhead for computing. The thermal model needed to be updated just a few times per minutes.

In order to better show how the proposed *proactive* thermal management, called PTM, keeps the given thermal constraint T_{safe}, we compared to another *reactive* thermal management policy [16], called RTM. In the RTM, it checks the current temperature every 1 second, and when the current temperature meets the given thermal constraint T_{safe}, it changes the frequency at one step lower level. In our evaluation, for a given 45 °C thermal constraint, we measured temperatures while executing the test apps for 5 minutes. We recorded the user interaction sequence using a custom user interaction recording tool, and replays it for both RTM and PTM. Figure 13 shows the temperature comparison result for the two management policies. The result shows that our proposed PTM more effectively keeps the thermal constraint compared to the RTM. While the RTM often violates the thermal constraint since the RTM can react only when exceeding the given T_{safe}, our proposed technique can change the frequency in advance based on the distinct workload characteristics of different user activities.

VI. Conclusion

In this paper, we proposed a novel automated user activity recognition technique which can be applicable for various optimizations. By carefully monitoring how each user interaction works in the smartphone systems, the proposed technique builds a hierarchical model which represents user activities. Based on the proposed technique, we also design a new type of power and thermal system management techniques which take into account the user activity phase changes. Our experiments show that the proposed power management technique can save CPU energy of applications by up to 28%, while the thermal management policy proactively keeps the thermal constraint at a safe level.

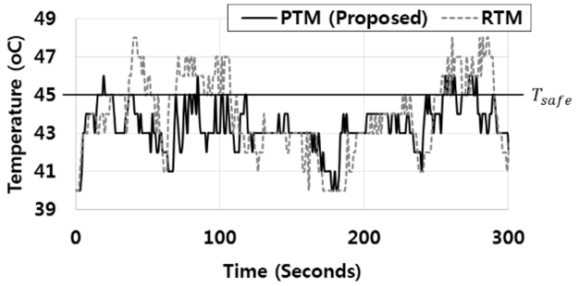

Fig. 13. The impact of the activity phase-aware thermal management policy on energy saving

Acknowledgment

This work has been supported by National Science Foundation (NSF) SHF grant 1218666 and NSF award 1344153.

References

[1] X. Li, G. Yan, Y. Han, and X. Li, "SmartCap: User Experience-Oriented Power Adaptation for Smartphone's Application Processor," DATE, 2013.
[2] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," ACM MobiSys, 2012.
[3] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," ACM MobiSys, 2010.
[4] A. Shye, B. Scholbrok, G. Memik, and P. A. Dinda, "Characterizing and modeling user activity on smartphones: summary," ACM SIGMETRICS, 2010.
[5] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," ACM MobiSys, 2011.
[6] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: multi-layer profiling of android applications," ACM MobiCom, 2012.
[7] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," USENIX OSDI, 2012.
[8] Android Developer, UI Thread, https://developer.android.com/training/multiple-threads/communicate-ui.html
[9] D. Defays, "An efficient algorithm for a complete link method," The Computer Journal, 1977.
[10] Wikipedia, Jaccard Index,http://en.wikipedia.org/wiki/Jaccard_index
[11] Microsoft Press, "Inside Microsoft .Net IL Assembler", 2002
[12] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "Lifetime Reliability: Toward an Architectural Solution" IEEE MICRO, 2005.
[13] M. Berhe, "Ergonomic Temperature Limits for Handheld Electronic Devices", ASME InterPACK, 2007.
[14] A. Coskun, T. Rosing, and K. Gross, "Utilizing Predictors for Efficient Thermal Management in Multiprocessor SoCs," IEEE Transactions on Computer-Aided Design, 2009.
[15] Y. Bai, "Memory Characterization to Analyze and Predict Multimedia Performance and Power in an Application Processor," White Paper Marvell, 2011.
[16] I. Rodero, E. Lee, D. Pompili, M. Parashar, M. Gamell, and R. Figueiredo, "Towards energy-efficient reactive thermal management in instrumented datacenters," IEEE/ACM GRID, 2010.
[17] Qualcomm, Snapdragon 8660 MDP, https://developer.qualcomm.com/snapdragon-mobile-development-platform-mdp
[18] Qualcomm, Trepn, http://developer.qualcomm.com/mobile-development/performance-tools/trepn-profiler
[19] H. Han, J. Yu, H. Zhu, Y. Chen, J. Yang, G. Xue, and M. Li, "E3: energy-efficient engine for frame rate adaptation on smartphones," ACM Sensys, 2013.
[20] Y. Kim, J. Kim, "Personalized Diapause: reducing radio energy consumption of smartphones by network-context aware dormancy predictions," USENIX HotPower, 2012.
[21] Wikepedia, "Student's t-test", http://en.wikipedia.org/wiki/Student%27s_t-test
[22] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," IEEE ICSE, 2013.