SmartCap: User Experience-Oriented Power Adaptation for Smartphone's Application Processor

Xueliang Li*†, Guihai Yan*, Yinhe Han *and Xiaowei Li*
*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
{lixueliang, yan_guihai, yinhes, lxw}@ict.ac.cn

***Abstract*—Power efficiency is increasingly critical to battery-powered smartphones. Given the using experience is most valued by the user, we propose that the power optimization should directly respect the user experience. We conduct a statistical sample survey and study the correlation among the user experience, the system runtime activities, and the minimal required frequency of an application processor. This study motivates an intelligent self-adaptive scheme, SmartCap, which automatically identifies the most power-efficient state of the application processor according to system activities. Compared to prior Linux power adaptation schemes, SmartCap can help save power from 11% to 84%, depending on applications, with little decline in user experience.**

I. Introduction

Nowadays people are enjoying the multi-functional smartphones such as Apple iPhone and Android Phones, but also annoyed by the gradual shrinking stand-by period. Even the state-of-the-art lithium batteries with capacity of kilo-milliampere hour energy can hardly sustain the power-hungry smartphones over 24 hours per recharge period under regular intensity of usage. Given the slowly increased battery capacity, we have to resort to developing more "economical" power usage to make our smartphones stand longer hours.

This paper focuses on application processors (APs) which play an increasingly important role in smartphones. AP is the key execution engine for various prevalent applications such as games, browsers, productivity tools. These applications consume even more power than the traditional communication components, making AP increasingly power-consuming.

User experience (UX) is the primary, if not the only, metric to evaluate smartphones; therefore when optimizing power usage for smartphones, we should directly respect to UX. This goal is different from traditionally theoretical peak performance or throughput which inclines to over-provision computational resources in smartphones, hence leading to poor power efficiency. In this paper, we find that a UX-oriented power management strategy will open a new power-saving opportunity.

We propose SmartCap, a novel scheme that only relies on the smartphone's runtime activities to infer the AP's proper power/performance state on which the power saving is achieved without compromising with UX. The runtime activities are obtained by fully taking advantage of the various sensors and system monitoring knobs built in current smartphones.

In particular, we make the following contributions:

- We quantitatively study the correlation between the runtime activity and UX, and discover the opportunity for power saving in smartphones.
- We build an inference model based on sample survey. This model is fed with the runtime statistics and yield the most power-efficient performance/power state of AP.
- With the model, we propose SmartCap scheme, a self-adaptive power management scheme. SmartCap does not need to interact with the users and all of the inputs are obtained from the existing sensors and system monitoring knobs.

The rest of the paper is organized as follows: Section II describes the study of UX, which motivates the SmartCap Approach proposed in Section III. Section IV shows the experimental setup and results. Section V presents the related work and followed by conclusion in Section VI.

The work was supported in part by National Basic Research Program of China (973) under grant No. 2011CB302503, in part by National Natural Science Foundation of China (NSFC) under grant No.(61100016, 61076037, 60921002).

II. The Study of User Experience

Although UX usually represents a subjective feeling [1][2], it is still measurable with statistical approaches. In this section, we first present the experimental platform and applications, then quantitatively characterize the UX based on a statistical sample survey, and finally introduce the motivation.

A. Experimental Setup

Platform: We use Motorola ME525 running Android 2.2 operating system (OS) as the experimental platform. The ME525 has a TI OMAP3610 chipset with an ARM Cortex A8 application processor equipped with Dynamic Frequency Scaling (DFS) supporting 300MHz, 600MHz, and 800MHz, a PowerVR SGX530 integrated GPU, and a 3.7 inch 16 million pixel TFT screen. Overall, this platform aims to represent the state-of-the-art smartphone configuration.

Applications: We choose 6 hot applications which intend to represent different usage scenarios. For example, `Talking Tom`, `Fruit Ninja`, and `Snow Pro` aim to represent three distinctive types of games, `Storm` to multimedia playing, `UC Browser` to web surfing, `QQ` to social network applications. For each application, we define a set of specific experience steps for the user to follow, thereby ensuring that every user fully exercise the applications.

B. Motivation: Sample Survey on User Experience

We polled 20 users on the satisfaction over a rang of applications under different processor frequency rate. During each testing session, we randomly shuffle the frequencies and do not leak the engaged frequency to the user. We classify the user experience into three ranks: Poor, Acceptable, and Good.

We first show the distribution of UX on six applications in Figure 1. Each bar denotes the number of users on the corresponding experience levels. Intuitively, we anticipate the proportion of users grading "Good" experience should increase with higher frequency, while that of the "Poor" should decrease, as `Talking Tom`, `Snow Pro`, and `Storm` show. However, some results clearly go against this intuition: the distribution of `Fruit Ninja` and `UC Browser`, for example, does not change much from 600MHz to 800Mhz. The similar situation also applies to `QQ` when frequency shifting from 300MHz to 600Mhz. We call the frequency beyond which the user experience is improved little as "saturated frequency". This phenomenon happens in most applications of our study. By identifying these saturated frequency rate, we can save power by avoiding the unnecessarily higher frequency.

For convenience in the rest of the paper, we define UX as Eq.(1) shows.

$$UX = \frac{w_1 \times N_1 + w_2 \times N_2 + w_3 \times N_3}{N_1 + N_2 + N_3}, \quad (1)$$

978-3-9815370-0-0/DATE13/©2013 EDAA

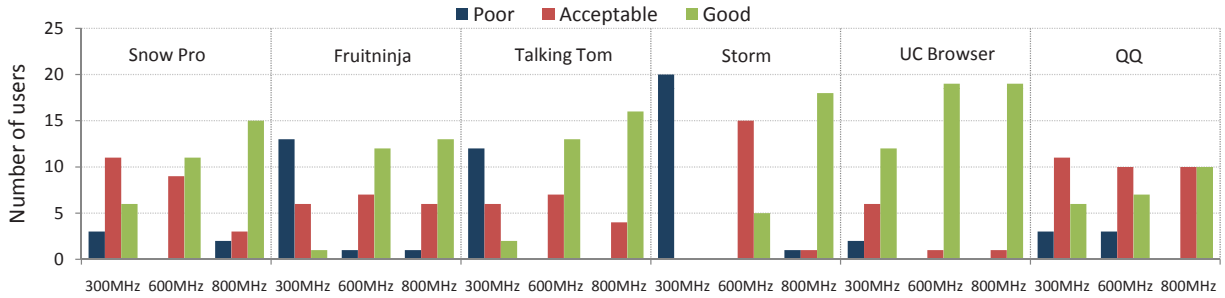

Fig. 1. User experience (UX) distribution on six applications.

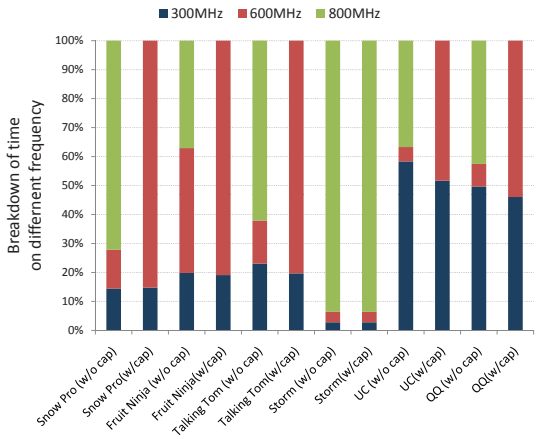

Fig. 2. Breakdown of time on different frequency with and without adjusting frequency cap. The percentages are calculated on average among different users

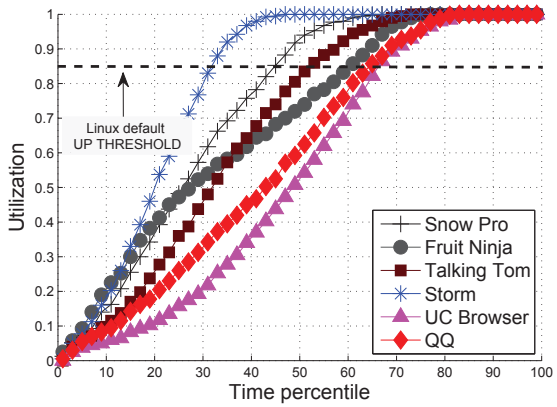

Fig. 3. CPU utilization. The x-axis is time percentile sorted in increasing order of utilization.

if $cpu\_util \geq$ `UP_THRESHOLD` **then**
  $Set\_Frequency$(`highest_frequency`)
else if $cpu\_util \leq$ `DOWN_THRESHOLD` **then**
  $requested\_freq \leftarrow$ *frequency that maintains a utilization at least* `UP_THRESHOLD` $- 10\%$
  $Set\_Frequency(requested\_freq)$
end if

Fig. 4. Linux ONDEMAND DFS algorithm

where, w_1, w_2, w_3 denote the weight for three levels of satisfactory (Poor, Acceptable, Good), and N_1, N_2, N_3 is the number of users grading the corresponding experience degree. In this paper, we assign w_1, w_2, w_3 to 1, 2, 3, respectively. So UX should be between 1 and 3, and higher UX reflects better user experience.

III. SmartCap Approach

Given a runtime session, the objective of SmartCap is to automatically figure out the saturated frequency, at which the UX is still close to that at highest frequency.

A. Rationale for Frequency-Capping

Ideal DFS operations should be able to automatically avoid frequency over-provisioning. However, the current Linux "ONDEMAND" DFS algorithm is far from ideal. The details of the algorithm is shown in Figure 4. We can see that the Linux ONDEMAND algorithm tunes the frequency according to only CPU utilization.

However, only relying on CPU utilization can not guide a power-efficient DFS scheme well. Take `UC Browser` for example, as shown in Figure 2, when the ONDEMAND algorithm [3] takes over the target system, `UC Browser` spends almost 40% of its time at the frequency of 800Mhz — a clearly over-provisioning as the sampling survey indicates in section II-B. However we tune `UP_THRESHOLD`, we cannot escape from the similar situation of power inefficiency. This phenomenon actually is not rare in our study. According to the trace of CPU utilization, as shown in Figure 3, we can infer that even though `UP_THRESHOLD` is set to 100%, the most aggressive mode, there is also nearly 25% time at 800Mhz for `UC Browser`.

We take another way, i.e. frequency-capping to eliminate frequency over-provisioning. This optimization can be readily integrated into the baseline Linux DFS algorithm. The key idea is to adaptively restricting the `highest_frequency`. Referring to `UC Browser`, setting the frequency cap at 600Mhz can totally avoid the over-provisioning of 800Mhz, and meanwhile won't compromise the time at 300MHz. As Figure 2 shows, we compare original (w/o cap) time breakdowns on each frequency with that under frequency cap (w/ cap). The results shows that the prohibition on over-provisioned frequency won't lead to time increase in 300MHz; hence the net benefit of power efficiency is guaranteed.

B. Correlating Frequency Cap and Features

SmartCap relies on the system activity features to infer the frequency cap. Together with CPU utilization discussed in Section III-A, we exploit several other CPU-related features to more faithfully capture the computation requirement for CPU. Also, we find these features do not always yield accurate inference (i.e. the predicted frequency cap of an application is equal to its saturated frequency); by combining other two usage features (i.e. touch features and phone movement features) the accuracy can be improved significantly.

1) CPU feature study: Let's start with the CPU-related features which turn out to be the primary indicators for application's CPU demand. Three features are exploited: 1) The ratio of Busy Time (R_{bt}) to total runtime, 2) the percentage of runtime at each frequency rate (P_f), and 3) the mean and deviation of lengths of Continuously Busy sessions ($\mu(Lcb)$, $\delta(Lcb)$), detailed as follows.

R_{bt} is the ratio between accumulated time when CPU utilization keeps at 100% to the total runtime interested. Larger R_{bt} usually implies more demand for computation capability. For example, as shown in Figure 3, the R_{bt} of `Storm` is as high as 65%, and the associated frequency cap is 800Mhz. By contrast, the R_{bt} of `QQ` and `UC` is as low as 30%; it turns out 600MHz frequency cap works well.

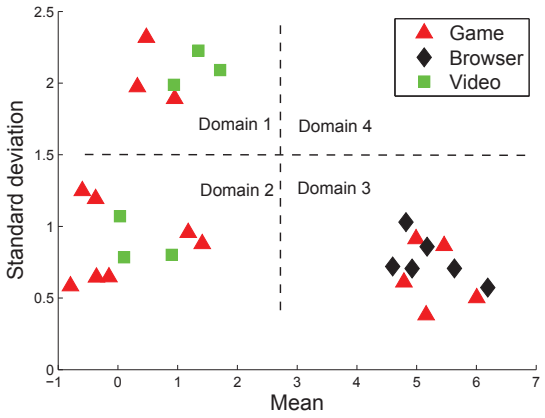

Fig. 5. Motion feature of 27 applications

However, only R_{bt} is not enough, because it is oblivious to the frequency. For example, different applications may have the same R_{bt}, but at different frequency; the application at higher frequency clearly is more computing intensive than that at lower frequency. Hence, another feature P_f is involved. As shown in Figure 2, under the original scheme(w/o cap), applications possess their own P_f values.

Continuously Busy session (CBsession) is the time slot when CPU utilization is continuously 100%. Usually a long-lasting and heavy workload will cause large CBsessions, which result in high $\mu(Lcb)$. By contrast, a sporadic workload can only lead to small CBsessions, which result in low $\mu(Lcb)$. An application with high $\mu(Lcb)$ should be serviced by a high CPU frequency to guarantee responsiveness. We use $\delta(Lcb)$ to further classify different applications with various CPU usage patterns.

While CPU-related features can correctly profile the demand for CPU performance in most cases, we find the inference accuracy is around 70%, still insufficient. This is probably because even if two applications share the similar CPU-related features, they indeed need different frequency caps. From machine learning fundamentals, we need to incorporate more features to differentiate these applications. In view of this, we exploit two more phone usage features, i.e. screen touch features and motion features, to further classify applications.

2) Usage feature study: The first usage feature comes from screen touches. Different types of applications usually own unique screen touch features reflected by the intensity of OS interrupt caused by screen touch. Applications' touches may show different touch time. For example, a slide over the screen is usually much longer than a click. The interval between touches also varies with different usage scenarios. Considering these cases, we extract three features from screen touch activity: 1) the mean and deviation of the numbers of interrupts (N_{int}) caused by touches ($\mu(N_{int})$, $\delta(N_{int})$), 2) the mean of touches's time length ($\mu(L_{touch})$), and 3) the mean and deviation of lengths of time intervals ($L_{interval}$)between touches ($\mu(L_{interval})$, $\delta(L_{interval})$).

The second usage feature comes from phone motion. Motion features are abstracted from the trace of gravity sensor, as shown in Figure 5. The X-axis is the mean value of gravity on vertical axis of the smartphones's coordinate system for a set of samples, denoted by ($\mu(y)$). The Y-axis is the standard deviation ($\delta(y)$). We plot the $\mu(y)$ and $\delta(y)$ of different applications in Figure 5. For clarity, applications are labeled by `game`, `browser` and `video`. The space is divided into four domains. Domain1 contains applications which need hold the phone horizontally and swap frequently, for example, racing games. Domain2 mainly corresponds to horizontal-holding applications with little swap. Some vertical-holding applications and with little swap motion, like browsers, are in Domain3. Surprisingly, no application locates in Domain4 in our experiments.

The results show that either using CPU features or usage features alone can only achieve around 70% inference accuracy. Combining them together, however, we can improve the accuracy up to 95%. This actually justifies the features exploited above.

3) Machine Learning Procedure: We use the above 12 features as input of our inference model. Our training model employs a three-layer neural network (NN): input layer with 12 neurons, hidden layer with eighteen neurons, and output layer with four neurons. We have tried out more neurons and more complex NN, which does not improve results but aggravate computation. The output layer produces a four-element binary vector corresponding frequency caps of 300MHz, 600MHz, 800MHz, and HIGHER, respectively. Note that if the output indicating the "HIGHER" means the application is so performance-hungry that even the highest 800MHz cannot satisfy its appetite. We first collected a large set of data from 20 users with 27 applications as the training set, then feed the information into a training process. Please refer [4] for more model training details.

TABLE I
THE PREDICTION DISTRIBUTION

Actual \ Inferred	300MHz	600MHz	800MHz	HIGHER
300MHz	**26.1%**	4.7%	0%	0%
600MHz	4.5%	**56.9%**	0.3%	0%
800MHz	0%	1.3%	**3.6%**	0%
HIGHER	0.5%	1.3%	0%	**0.8%**

We implemented a 4-fold cross validation procedure to validate our model. The raw data is divided into into four groups and we randomly choose three groups as training set, the other one as validation set. We obtain a average training set accuracy at 92% and cross validation set accuracy at 80%. We also use five new applications which are neither appearing in training set or cross validation set to test the well-trained model. Result shows 82% accuracy can be achieved.

4) SmartCap Implementation: The SmartCap is implemented into a client-server style application. The server is responsible for training the model, then transmits the inference parameters to clients, i.e. the user side smartphone. At user side, SmartCap builds a history table to record each application's frequency cap obtained from the inference model at its first run. Therefore, for each application, the inference procedure needs to run only once. When an application is evoked, the OS first index the history table. If it turns out to be a miss, then the OS initiates an inference procedure and append a new frequency cap entry to the table for the new application.

Concerning the advent of new types of applications never seen before, SmartCap should adaptively improve itself. Therefore, we provide user interface (UI) to manually adjust individual applications' frequency cap if need. Meanwhile, our background service will collect this frequency cap value and the trace of system activity, and then submit to our server-side. These data eventually become our new training example for machine learning in order to calibrate our model. Thus, we can constantly improve our inference model.

IV. Experiments

A. Experimental setup

Our experimental target is Motorola ME525 running Android 2.2 operating system. We use PowerMonitor [5] to measure realtime power consumption of mobile device. The sample frequency is 5KHz with 0.01 milliwatt resolution. We collected detailed activity traces from 20 users using 27 applications, most of which are top popular ones on Google Play [6]. The total testing time is as long as 20 hours. Each testing session lasts for about 3 minutes, as we described in Section II-B.

B. Inference Accuracy

We define frequency cap as the lowest frequency at which UX is within 0.3 less than that at highest frequency. The inference accuracy for frequency cap is crucial to avoid unexpected UX decline. We therefore first evaluate the inference accuracy. The results shows in Table I, where the diagonal element represents the percentage of correct inferences, which takes up 87.4% of all inferences. The elements in upper triangular matrix show false positive inferences which will lead to loss of opportunity for power saving, but won't hurt the UX; while the elements in lower triangular matrix show false negative which underrates the frequency cap required, thereby degrading UX. Overall, the false positive only take up 5%, and false negative 7.6%. From this result we conclude that in 92.4% scenarios SmartCap won't hurt the UX, so we can safely use it.

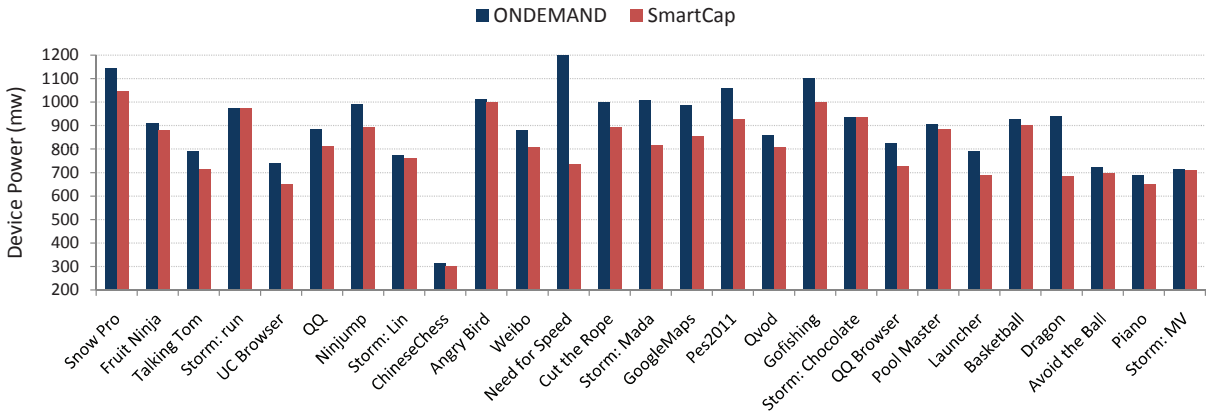

Fig. 6. Power consuming of the entire device.

C. Power Saving

We first study the AP's power saving. We use PowerMonitor to trace the power consumption of each application for 5 minutes. SmartCap meets the performance of oracle scheme, and power saving ranges from 11% to 84%, depending on applications.

Generally, the power saving for most computing-non-intensive applications such as `Storm:Lin` and `Chinese Chess` is very significant, but for computing-intensive scenarios such as `Storm:Run` and `Gofishing` is not that much. However, computing-intensity is not always reliable indicator for power saving. Take `Need for Speed` for example, it's computing-intensive but surprisingly with a low frequency cap. Fortunately, SmartCap is able to accurately identify these occasions, as the results show. This result proves that SmartCap can reliably target a wide range of power-saving scenarios.

The power saving on AP can contribute to the system-level power saving, which is decided by two factors: 1) the proportion of AP's power to the whole device's power, and 2) the relative reduction on AP's power. The results are shown in Figure 6, where we compare the "Original" scheme, i.e. the linux ONDEMAND DFS strategy, with the SmartCap-enabled DFS strategy. The result shows that system-level power saving ranges from 10 mW to 466 mW with an average value of 82mW. When the AP dominates the entire system power and also shows high relative power, such as `Need for Speed` and `Storm:Dragon`, large system-level power reduction is expected: they each achieves $466mW$ and $256mW$ power reduction. Particularly, for some applications such as `UC Browser` and `QQ`, although the AP's power saving is not that very significant, it also contributes to 92 mW and 73 mW power reduction to the entire device. This is because the AP's basic power consuming dominate the system's power when running the these applications.

This tens of milliwatt system-level power reduction is meaningful for smartphone, because it can greatly improve the standby time. Our smartphone's power at idle state is as small as 25mW, and SmartCap's average power saving at active state is about 82mW. This means that if the aggregated runtime of the smartphone for five hours, for example, SmartCap can help extend the standby time as long as 16 hours.

V. Related Work

The optimization for smartphones becomes increasingly hot topic with the prevalence of smartphones [7][8][9][10]. Unlike studies focus on power-aware network interface and location sensing [11][12][13] and display [14], our work focuses on another power-hungry component: application processor. There are also many studies on low-power CPU design techniques, such as DFS [15][16] and heterogeneous architecture [17][18]. However, these work either does not target the mobile devices or or do not directly respect the UX. Our work fully uses the characteristics of sensor-rich and smartphone applications and users, therefore is different from prior work.

VI. Conclusions

We proposed SmartCap, a novel scheme to avoid smartphone's AP frequency over-provisioning. We found in some usage scenarios, the higher frequency does not yield appreciable user experience improvement, hence should be avoided. We developed an inference model based on neural network to accomplish the minimal frequency inference. Experimental Results show that SmartCap can save AP power from 11% to 84%, depending on applications, with little compromising with UX.

References

[1] E. Law, V. Roto, A. P.O.S. Vermeeren, J. Kort, M. Hassenzahl, "Towards a shared definition of user experience," *CHI*, pp. 13–24, 2008.

[2] M. Hassenzahl, N. Tractinsky, "User experience - a research agenda," *Behaviour & Information Technology*, pp. 91–97, 2006.

[3] A. Shye., B. Scholbrock, G. Memik, Y. Pan, J. S. Miller, P. A. Dinda, R. P. Dick, "Power to the People: Leveraging Human Physiological Traits to Control Microprocessor Frequency," *Micro*, pp. 188–199, 2008.

[4] T. M. Mitchell, "Machine Learning," *McGraw-Hill Higher Education; New edition edition*, pp. 81–126, 1997.

[5] Power Monitor, "http://www.msoon.com/LabEquipment/PowerMonitor/,"

[6] Google Play, "https://play.google.com/store,"

[7] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos.,R. Govindan, D. Estrin, "Diversity in Smartphone Usage," *MobiSys*, pp. 179–194, 2010.

[8] T. M. Mitchell, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," *Micro*, pp. 168–178, 2009.

[9] Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," *CODES/ISSS*, pp. 105–144, 2010.

[10] R. Rar, S. Vrudhula, D.N. Rakhmatov, "Battery modeling for energy aware system design," *Computer*, vol. 4, no. 4, pp. 77–87, 2003.

[11] Z. Zhuang, K. Kim, J. P. Singh, " Efficiency of Location Sensing on Smartphones," *MobiSys*, pp. 315–330, 2010.

[12] F. R. Dogar, P. Steenkiste K. Papagiannaki, " Catnap: Exploiting High Bandwidth Wireless Interfaces to Save Energy for Mobile Devices," *MobiSys*, pp. 107–122, 2010.

[13] M. R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, M. J. Neely, "M. R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, M. J. Neely," *MobiSys*, pp. 255–270, 2010.

[14] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, R. K. Balan, " Adaptive Display Power Management for Mobile Games," *MobiSys*, pp. 57–70, 2010.

[15] A. Sinha, A. P. Chandrakasan, "Dynamic Voltage Scheduling Using Adaptive Filtering of Workload Traces," *VLSI Design*, pp. 221–226, 2001.

[16] A. Mallik, B. Lin, G. Memik, P. Dinda, R. P. Dick, "User-Driven Frequency Scaling," *Computer Architecture Letters*, p. 16, 2006.

[17] F. X. Lin, Z. Wang, R. LiKamWa, L. Zhong, "Reflex: using low-power processors in smartphones without knowing them," *ASPLOS*, pp. 13–24, 2012.

[18] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor and S. Swanson, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro*, pp. 86–95, 2011.