Reducing Energy Consumption of Smartphones Using User-Perceived Response Time Analysis

Wook Song, Nosub Sung, Byung-Gon Chun[†], and Jihong Kim
Department of Computer Science and Engineering
Seoul National University, Korea
{wooksong,nssung,jihong}@davinci.snu.ac.kr
[†]{bgchun}@snu.ac.kr

ABSTRACT

We propose a novel power optimization framework based on user-perceived response time analysis. Unlike most existing power optimization approaches, our framework takes explicit account of the quality of user experience into applying low-power techniques. We divide an execution of a given user-interactive session into two intervals, one where the system response time directly affects user experience and the other where the system response time does not affect user experience. For the user-oblivious response time interval, our framework allows more aggressive applications of low-power techniques for a higher energy efficiency. In order to identify the user-perceived response time of smartphone applications during run time, we developed an on-line **u**ser-perceived **r**esponse time **a**nalyzer (**ura**) for Android-based smartphones. Based on the analysis result of **ura**, our proposed power optimization framework employs more aggressive low-power techniques in the user-oblivious interval. Our experimental results on a smartphone development board show that the proposed technique can reduce the CPU energy consumption by up to 65.6% over the Android's default `ondemand` *cpufreq* governor.

1. INTRODUCTION

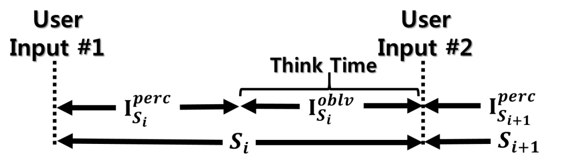
Smartphones are highly interaction-oriented devices because most of the usage scenarios on smartphones involve frequent user interactions with smartphones. Moreover, in most cases, a user tends to focus on one app[1] at a time although multi-tasking support is commonly available for modern smartphones [1]. For example, when reading new incoming emails, the user first launches an email app. Once the email app is launched, the user opens an email and reads it for a while. In this example, the user's whole attention is directed to a single interactive session at a time. Therefore, the quality of user experience with smartphones largely depends on how smoothly and quickly smartphones react to the user's various interactions. In the case of an email app, the quality of user experience is significantly affected by the launching time (which can be regarded as the response time of an app launching interaction) and the response time of loading a selected email. Since the response time of an interactive session has a large impact on the quality of user experience, understanding and analyzing the response time of an interactive session in smartphones are important requirements for improving user experience.

[1]In this paper, we call an application by a (more popular) shorthand, app.

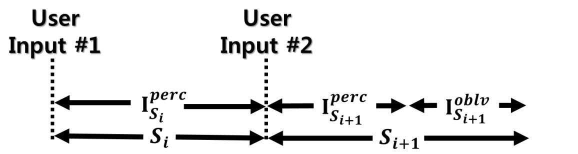
In conventional computing systems, the response time of a task is defined as the length of the time interval between the start and the end of a task execution. However, for smartphones, most users have a tendency to interact with the smartphones in a hurried fashion so that they *subjectively* decide that the smartphones are ready for the next interaction even though the task execution has not been fully completed. For this reason, the existing definition of the response time, which we call *computation-centric*, is not appropriate for accurately representing the user-perceived effective response time in smartphones [2]. For example, in the case of launching an email app, a user may consider that the launching has been completed when the visible user interface for the next interaction appeared on the display as its final form even though several display-insensitive computations may be still executing. In this case, the *user-perceived* response time is a lot shorter than the computation-centric response time. Therefore, in order to accurately represent the user-perceived response time in smartphones, we need a different definition of the response time from the smartphone user's perspective.

In this paper, we propose such a definition of the response time, which we call *the display-centric response time*, which is known to be a critical metric for the quality of user experience of the smartphone. The display-centric response time is defined as the period from the beginning of a user interaction (e.g., touch) to the time when all of the visible interface for the next interaction is drawn, which indicates the *effective* response time felt by users. (In this paper, we use terms the display-centric response time and the user-perceived response time, interchangeably.) The display-centric response time distinguishes the task execution in two parts, one affecting the visible portion of the display (called a display-sensitive part) and the other not affecting the visible portion of the display (called a display-insensitive part). For example, in the case of an email app, tasks such as loading cached emails from the storage and fetching new emails from the servers are examples of the former. Background tasks such as downloading attached files and embedded images are examples of the latter.

If we can identify the end of the display-sensitive part of a task execution (that is, the display-centric response time) during run time, more intelligent system optimizations can be applied to smartphones. For example, if an OS knew that the current execution does not affect the user visible display



(a) The case when S_i is divided into $I^{perc}_{S_i}$ and $I^{oblv}_{S_i}$.



(b) The case when S_i consists of $I^{perc}_{S_i}$ only without any think time.

Figure 1: Two cases on how S_i is divided into $I^{perc}_{S_i}$ and $I^{oblv}_{S_i}$.

for the next interaction, an OS power management scheme may lower the CPU clock frequency more aggressively for a higher energy efficiency without any negative effect on user experience. Existing mobile OS power management schemes (such as the *cpufreq* governors of the Linux kernel), however, cannot make such aggressive decisions because the CPU frequency is mostly decided by the recent CPU utilization history which cannot quickly react to changing contributions of the current computation on the quality of user experience.

In this paper, we propose a novel CPU power management framework based on on-line user-perceived response time analysis. In order to identify the display-centric response time of smartphone apps during run time, we developed **ura**, a **u**ser-perceived **r**esponse time **a**nalyzer for Android-based smartphones. Based on **ura**'s on-line identification of the display-centric response time, our proposed framework enables more aggressive low-power techniques to be employed while executing display-insensitive parts of task executions (which do not affect the user-perceived response time). As a concrete example of low-power techniques, we use dynamic voltage scaling (DVS) to demonstrate our proposed framework.

In order to evaluate our proposed technique, we implemented **ura** and a **ura**-based CPU power management governor in the Android platform, version 4.0.4 (ICS) running on the Samsung Exynos 4x12-based SMDK smartphone development board. Our SMDK board is a specialized development board for Galaxy S3 smartphones with on-board component-level power measurement support. Experimental results show that the proposed technique can reduce the CPU energy consumption by up to 63.8% over the Android's default `ondemand` policy without degrading the quality of user experience.

The rest of this paper is organized as follows. We explain the key idea behind our proposed framework in Sec. 2. In Sec. 3, we describe an overview of **ura** and illustrate how the **ura**-based CPU power management technique can improve the CPU energy efficiency. We report experimental results in Sec. 4. Sec. 5 presents related work and Sec. 6 concludes with a summary and future work.

2. BASIC IDEA

As a truly interaction-oriented device, most usage scenarios of a smartphone are composed of a sequence of *interactive sessions*, S_1, ..., S_N. Each interactive session S_i is defined as an interval between two consecutive user inputs. We can further divide the execution of an interactive session S_i into two subintervals, $I^{perc}_{S_i}$ and $I^{oblv}_{S_i}$, a *user-perceived response time interval* and a *user-oblivious response time interval*, respectively. $I^{perc}_{S_i}$ represents the period from the beginning of the interactive session S_i initiated by a certain user input to the time when all of the user-visible interface for the next user interaction are displayed. (In other words, the length of $I^{perc}_{S_i}$ is the display-centric response time of S_i.) $I^{oblv}_{S_i}$, on the other hand, corresponds to the user's think time before the next user interaction. (The length of $I^{oblv}_{S_i}$ is determined by the time when the next user input is entered for the next interaction.) Figure 1 illustrates how the interactive session S_i may be further divided into $I^{perc}_{S_i}$ and $I^{oblv}_{S_i}$. In the case of the example in Figure 1(a), after all the user-visible contents are drawn for the user input #1, S_{i+1} is initiated after some think time (i.e., after $I^{oblv}_{S_i}$) by the user input #2. On the other hand, the user may input the user input #2 right after $I^{perc}_{S_i}$ without any think time.[2] Figure 1(b) shows such an example. The length of $I^{oblv}_{S_i}$ is almost zero in this case and the new interactive session S_{i+1} begins without any think time. Since the system performance level in I^{oblv}_{S} is less likely to affect the quality of user experience, we may take a more aggressive approach in optimizing power/energy consumption while executing in I^{oblv}_{S} without degrading the quality of user experience.

In order to better motivate our proposed optimization framework, we illustrate how the energy consumption of an app launching interactive session S_L can be improved using an example. Figure 2 shows how the CPU power consumption changes during the first 15 seconds after the Android `web browser` is launched. In order to emphasize differences between $I^{perc}_{S_L}$ and $I^{oblv}_{S_L}$, we chose the mobile start page of Yahoo (which included a fair amount of background computations in $I^{oblv}_{S_L}$). The X-axis, the Y-axis on the left side, and the Y-axis on the right side represent the elapsed time, CPU frequency, and CPU power consumption, respectively. In this example, at t = 0.4, the `web browser` app is launched. After the required resources are downloaded from the Yahoo website, all the user-visible contents are drawn at t = 5.6. That is, $I^{perc}_{S_L}$=[0.4, 5.6]. As shown in Figure 2, although there are additional computations related to the launching session, the user perceives that the launching of `web browser` was completed at t = 5.6. (We will describe how to identify the end of $I^{perc}_{S_L}$ in Sec. 3.1.) Even when executing in $I^{oblv}_{S_L}$, we observe that the CPU frequency is frequently increased to the maximum frequency of 1,400 MHz. Furthermore, the

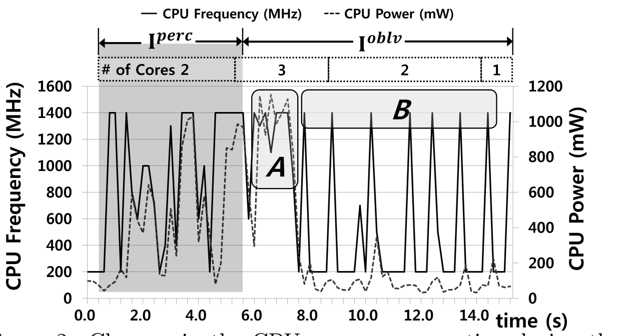


Figure 2: Changes in the CPU power consumption during the app launching session.

[2]There is the third case when the user input #2 is initiated within $I^{perc}_{S_i}$. Since our proposed technique does not change the execution behavior in $I^{perc}_{S_i}$, it is considered as the beginning of the new interactive session S_{i+1}.

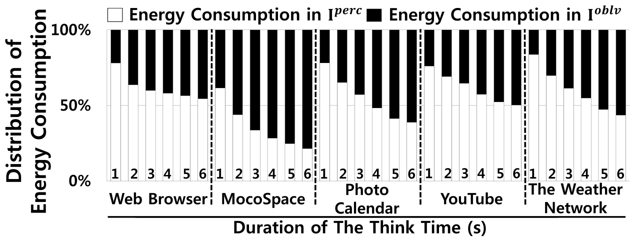


Figure 3: CPU energy consumption breakdowns between I^{perc} and I^{oblv} over varying durations of the think time.

number of active cores remains three[3] until t = 8.6. Since the execution in $I^{oblv}_{S_L}$ does not affect the quality of user experience, if we knew that the current execution were in $I^{oblv}_{S_L}$, we could have lowered the CPU frequency to the minimum frequency of 200 MHz.[4] In this case, the energy consumption in $I^{oblv}_{S_L}$=[5.6, 8.6] could be reduced by 47.0% over the Android's default CPU DVS policy, assuming that the next user input event occurred after the think time of 3 seconds.

In order to evaluate the applicability and effectiveness of our proposed framework over different apps, we measured the energy consumed in both I^{perc} and I^{oblv}, varying duration of the think time in our SMDK measurement board. Figure 3 summarizes breakdowns of the CPU energy consumption between I^{perc} and I^{oblv} for five apps over different durations of the think time. We measured the CPU energy consumption during the launching sessions of the five apps. Our measurements show that up to 78.7% of the total CPU energy is consumed in I^{oblv}. For example, in the case of `web browser`, when the think time is set to 6 seconds, 45.7% of the total CPU energy is consumed in I^{oblv}. When the think time is set to 3 seconds, 46.6% of the total CPU energy consumption is from I^{oblv} on average. Since we can lower the CPU frequency more aggressively in I^{oblv}, our measurement result strongly indicates that our proposed framework can significantly reduce the energy consumption of smartphones.

3. URA-BASED POWER OPTIMIZATION

3.1 User-Perceived Response Time Analyzer

Our **ura**-based power optimization framework requires the user-perceived response time of each interactive session S_i to be computed on-line. In the proposed framework, **ura** is responsible for the identification of the end of $I^{perc}_{S_i}$ during run time from the execution of S_i. Our proposed aggressive optimization technique, therefore, can be immediately applied from the first $I^{oblv}_{S_i}$ execution. In this section, we describe the design and implementation of our **ura**-based power optimization framework for the Android platform.

In Android, only one UI thread per app is allowed to update all the user-visible contents of an app. Furthermore, when a display-update request is issued by the UI thread, the Android platform does not immediately redraw the visible user interface. Instead, it is first posted to the event queue of the UI thread of the app and then, the UI thread subsequently dequeues the request and handles it. Exploiting Android's display update mechanism, **ura** can identify the end of $I^{perc}_{S_i}$ by tracking all the display-update requests related to the user interaction in a given interactive session S_i and detecting when the last display-update request of S_i is processed. Although the UI interaction with the user is the main source of generating display-update requests, it is not the only source. For example, in order to refresh user visible contents such as advertisement banners, display-update requests can be also generated. In this case, **ura** can automatically distinguish such requests from the display-update requests related to the user interaction because **ura** only tracks the display-update requests issued by the user input.

In order to detect the end of $I^{perc}_{S_i}$ for a given S_i, **ura** works as follows:

Step 1. Catch an interactive user input which indicates the beginning of S_i.

Step 2. Keep track of all the spawned threads from the user input (if any).

Step 3. Detect display-update requests related to serving the user input.

Step 4. Check whether all the display-update requests were processed so that the end of $I^{perc}_{S_i}$ can be decided. (In the case of when the next user interaction occurs before all the display-update requests are processed, the end of $I^{perc}_{S_i}$ can be also decided at this moment.)

Figure 4 shows an architectural overview of **ura** and the **ura**-based *cpufreq* governor within the Android platform. **ura** consists of two main modules, the modified method call interpreter, modInterpreter, and the end of user-perceived response time identifier, endIdentifier. As an additional module to the Dalvik VM, modInterpreter is responsible for steps 1, 2, and 3. For step 4, endIdentifier determines the user-perceived response time. By taking advantage of the user-perceived response time, the **ura**-based governor adjusts the CPU frequency to achieve a higher CPU energy efficiency. In the current implementation, the Davik VM interpreter is modified to instrument the method invocation and method return during run time. For this reason, **ura** cannot trace the native method invocations, which are included in native libraries such as the native OpenGL [5]. However, the OpenGL ES API, which is based on the Java language and provided by the Android SDK, are fully supported by **ura**.

ModInterpreter consists of three submodules, the input event detector, the spawned thread tracker, and the UI update detector. The main function of the input event detector is to capture events related to a particular user input. The spawned thread tracker is responsible for tracing newly spawned threads while processing the user input. All the message exchanges between the main thread and spawned threads are also traced by the spawned thread tracker. Besides, if the main thread sends messages to the other threads (which are already spawned before the user input), such threads are tracked as well. The UI update detector keeps track of display-update requests created for serving the user input.

Figure 5 illustrates how **ura** identifies the user-perceived response time using an example. When a user interacts with Android UI components such as WIDGET and VIEW packages, a callback method in the event listener interface is invoked to handle a particular interaction. (In order to support different types of user interactions, the Android SDK provides various callback methods. For example, user interactions such as a touch, a click, and a long-click are handled by

[3]The SMDK board has a quad-core ARM cortex-A9 as a main CPU.

[4]In Figure 2, `web browser` executes tasks such as storing the downloaded resources in the web cache, capturing a snapshot of the current web page, and updating the browser history (as shown in the area A.) In addition, in the area B, `web browser` regularly renders Javascript code in the web page even though the web page does not dynamically change the displayed contents [3].

[5]Although there are important apps (such as game apps) that use the native OpenGL, most Android apps are written in JAVA only.

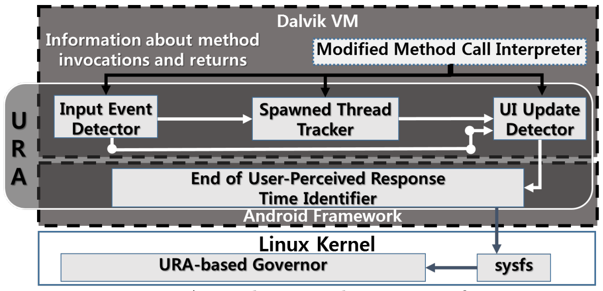

Dalvik VM
Information about method invocations and returns
Modified Method Call Interpreter
URA
Input Event Detector
Spawned Thread Tracker
UI Update Detector
End of User-Perceived Response Time Identifier
Android Framework
Linux Kernel
URA-based Governor
sysfs

Figure 4: An architectural overview of **ura**.

onTouch(), *onClick()*, and *onLongClick()* methods, respectively.) In the example in Figure 5, the callback method, *onClick()*, is called because the user clicks the user interface resource such as the BUTTON WIDGET. As the first step of identifying the user-perceived response time, the input event detector traces all the method invocations related to the callbacks for the user input, so as to identify the start t_s of the current interactive session S_i. In the case of the example in Figure 5, the input event detector catches the *onClick()* invocation. For steps 2 and 3, the input event detector also provides both the spawned thread tracker and the UI update detector with information about all the method invocations during the execution of *onClick()*.

When an app is launched, a special thread, called main thread, is created by the Android system. While only main thread can update the user-visible contents, compute-intensive work is performed by separate threads, called worker threads, for better responsiveness. If a worker thread requires updating the user interface, such requests are delegated to main thread. In order to support inter-process communication (IPC) between main thread and worker thread, various APIs are supported by the Android SDK, for interchanging *Message* and *Runnable* objects between the main thread and the worker thread. By exploiting the information (which is provided by the input event detector) on the method invocations during the execution of the callbacks for the user input, the spawned thread tracker traces newly spawned worker threads and all the invocations among such IPC APIs. For example, as shown in step 2 of Figure 5, when main thread wants to perform compute-intensive work via worker thread, main thread invokes *sendMessage()* while the worker thread invokes *dispatchMessage()*. In this case, the spawned thread tracker catches the *sendMessage()* and *dispatchMessage()* invocations. And then, in order to detect UI update requests created by worker thread, information about all the method invocations during the execution of *dispatchMessage()* is fed to the UI update detector.

To recognize the changes in the user-visible contents, **ura** traces UI update requests issued by the user input and captures the moment at which the last request is handled. For example, at step 3 in Figure 5, the *invalidate()* methods are invoked twice during the execution of both *onClick()* and *dispatchMessage()*. At these points, the UI update requests are posted to the event queue of main thread. In order to track the UI update requests, the UI update detector thus catches the *invalidate()* invocations and watches the event queue for the UI update requests. Subsequently, when main thread dequeues the last update request from the event queue and invokes *draw()* to handle it (at t_e in step 4 of Figure 5), endIdentifier determines t_e as the end of I^{perc}. In this example, the user-perceived response time is estimated as (t_e-t_s).

3.2 URA-based CPU Frequency Governor

Taking advantage of **ura**'s on-line identification of I^{oblv} for a given interactive session, we developed a new *cpufreq* governor, the `oninterval` *cpufreq* governor, for Linux CPU

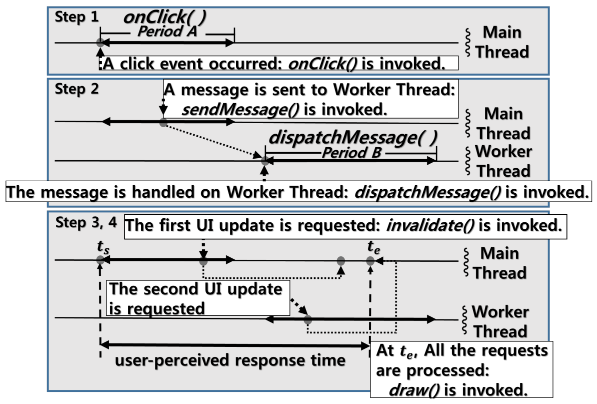

Step 1
onClick( )
Period A
A click event occurred: onClick() is invoked.
Main Thread
Step 2
A message is sent to Worker Thread: sendMessage() is invoked.
dispatchMessage( )
Period B
Main Thread
Worker Thread
The message is handled on Worker Thread: dispatchMessage() is invoked.
Step 3, 4
The first UI update is requested: invalidate() is invoked.
t_s
t_e
Main Thread
The second UI update is requested
Worker Thread
user-perceived response time
At t_e, All the requests are processed: draw() is invoked.

Figure 5: An example of identifying the user-perceived response time.

power management. Algorithm 1 describes how the `oninterval` *cpufreq* governor decides the CPU frequency. As with other Linux *cpufreq* governors, the CPU frequency is updated at each sampling period (e.g., 20 ms). The `oninterval` governor relies on endIdentifier of **ura** for keeping track of whether the current execution is in I^{perc} or I^{oblv}, as described on line 2 in Algorithm 1. Whenever the new interactive session S_i is started, the current execution interval type is set to I^{perc}. When endIdentifier detects the end of I^{perc}, it is changed to I^{oblv}. Based on this information, the `oninterval` governor employs the lowest CPU frequency while executing I^{oblv}. Furthermore, the number of active cores is also restricted to one for further power reduction in I^{oblv}. Otherwise, decisions by the `ondemand` *cpufreq* governor [4], which is the default governor in most kernels for the Android Open Source Project, are applied in adjusting the CPU frequency. When the CPU utilization exceeds the predefined upper threshold (e.g., 95, U_{high} in Algorithm 1), for higher responsiveness, the `ondemand` *cpufreq* governor quickly switches to the maximum CPU frequency. On the other hand, if the CPU is less loaded (e.g., when the CPU utilization falls below 20, U_{low} in Algorithm 1), the governor gradually decreases the frequency. Therefore, when the new user input is initiated, the `oninterval` *cpufreq* governor can rapidly adapt to changing CPU utilizations.

4. EXPERIMENTAL RESULTS

In order to evaluate the effectiveness of our proposed framework, we have implemented **ura** and **ura**-based CPU power management technique on the Samsung Exynos 4x12-based SMDK board running Android 4.0.4 (Ice Cream Sandwich). We modified the Dalvik VM interpreter for tracking all the method calls related to the identification of the end of the user-perceived response time. ModInterpreter and endIdentifier (which were described in Sec. 3.1) were implemented in the Dalvik VM and the Android Framework, respectively. We also modified the Linux kernel's *sysfs* interface slightly to support the **ura**-based CPU frequency governor. The `oninterval` *cpufreq* governor was also added to the Linux kernel, version 3.0.15. In our evaluations, we have experimented with 7 apps under different usage scenarios. Each app usage scenario consists of two consecutive interactive sessions. Table 1 summarizes selected apps and their usage scenarios. An in-house scenario replay tool, which was implemented using the MonkeyRunner tool [5], is used to automatically execute the usage scenarios.

Prior to evaluating the efficiency of the proposed **ura**-based CPU power management framework, we first validated if **ura** can accurately estimate the user-perceived re-

Algorithm 1 Pseudo code for the `oninterval` algorithm.

```
1: begin
2:    curExecIntervalType := getCurExecIntervalType();
3:    nextFreq := computeNextFreq_OnInterval(
                      curExecIntervalType);
4:    setCpuFreq(nextFreq);
5: end

6: function CpuFreq computeNextFreq_OnInterval(
                      curExecIntervalType)
7: begin
8:    if curExecIntervalType = $I^{perc}$ then
9:       curFreq := getCurFreq();
10:      return computeNextFreq_OnDemand(curFreq);
11:   else if curExecIntervalType = $I^{oblv}$ then
12:      //$f_{min}$ is the minimum CPU frequency.
13:      return $f_{min}$;
14:   endIf
15: end

16: function CpuFreq computeNextFreq_OnDemand(
                      curFreq)
17: begin //The ondemand algorithm starts from this point.
18:   curUtil := getCpuUtilzation()
19:   if curUtil >$U_{high}$ then
20:      //$f_{max}$ is the maximum CPU frequency.
21:      return $f_{max}$;
22:   else if curUtil <$U_{low}$ then
23:      nextFreq := curFreq × 0.8;
24:      return nextFreq;
25:   else
26:      return curFreq;
27: end
```

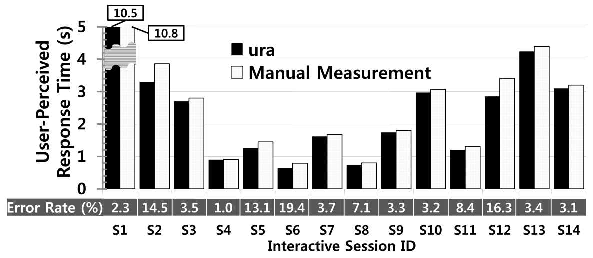


Figure 6: User-perceived response time differences between **ura** and manual measurements.

sponse time. In order to evaluate the accuracy of the estimated response time from **ura**, we have manually measured display-centric response times of our benchmark apps. For the manual measurement, we recorded the screen of the SMDK smartphone development board during the execution of each usage scenario with a digital video camera, which supports 30 fps frame rate. The recorded video was then analyzed frame by frame so that we can manually quantify the response time of the interactive sessions. Figure 6 compares user-perceived response times from manual measurement and **ura**. The X-axis and the Y-axis denote various Android apps and their user-perceived response times, respectively. As shown in Figure 6, **ura** accurately estimates the user-perceived response times with an average error of 5.2% over manually measured times, thus achieving a sufficient accuracy for **ura**-based power/energy optimizations. Moreover, in our implementation on the smartphone development board, **ura** incurs additional computation overhead by up to only 1.2% of the user-perceived response time whenever the user-perceived response time is estimated.

Table 1: Scenario descriptions of 7 benchmark apps.

App Name (Category)	Interactive Session ID	Interactive Session Description
News Republic	S1	Launching
(News)	S2	Viewing a list of all news
MocoSpace	S3	Launching
(Social Networking)	S4	Viewing a profile page
Photo Calendar	S5	Launching
(Photography)	S6	Selecting a photo album
Seesmic	S7	Launching
(Social Networking)	S8	Reading an article on Facebook
The Weather Network	S9	Launching
(Weather)	S10	Viewing a day info page
gReader	S11	Launching
(RSS Feed Reader)	S12	Reading a RSS feed
Web Browser	S13	Launching
(Web Browser)	S14	Clicking a link to an article

Figure 7 shows the impact of the proposed `oninterval` governor on energy savings over varying durations of the think time for 14 interactive sessions. The result shows that the `oninterval` governor can save the CPU energy on average by 27.0% over the `ondemand` governor when the duration of the think time is 3 seconds. For S12 (gReader), the maximum energy saving of 65.6% is achieved with 5 seconds of the think time. For 9 out of 14 scenarios, the energy saving ratios increase as the duration of the think time grows. For S7, S9, S11, S13, and S14, on the other hand, the percentage of energy savings decrease as the duration of the think time grows after those intensive computation periods. This is because most background computations are completed within a couple of seconds (2, 1, 1, 3, and 1, respectively) after the end of the user-perceived response time. As shown in Figure 7, although the maximum energy savings were observed when the duration of the think time is 6 seconds, the `oninterval` governor can be useful even under shorter think times. When the think time is decreased to 1 and 2 seconds, our proposed `oninterval` governor can save the energy consumption, on average, by 21.7% and 25.7%, respectively over `ondemand`. Since the CPU power can account for up to 40% of the total power consumption in the latest smartphones [6], the `oninterval` governor can reduce the total power consumption by up to 10.3% when the duration of the think time is 2 seconds.

In order to understand the impact of an aggressive DVS decision on the quality of user experience in the following interactive session, we compared, for each app usage scenario in Table 1, how the user-perceived response time of the second interactive session changes under the `oninterval` governor while varying the duration of the think time of the first interactive session. Table 2 shows normalized user-perceived response times of seven second interactive sessions, where user-perceived response times of seven second interactive sessions under the `ondemand` governor are used as baselines. We ran each scenario 100 times and the average of measured times was used for a comparison. For 5 out of 7 scenarios, normalized user-perceived response times range from 0.99 to 1.01. For S10 and S14, the `oninterval` governor increases the normalized user-perceived response time by up to 3% over the `ondemand` governor. Since our app usage scenarios all access remote servers through the wireless network (whose latency often fluctuates), we conclude that there is no significant difference in the user-perceived response time of the second interactive sessions between the `ondemand` governor and `oninterval` governor.

5. RELATED WORK

Recent investigations such as AppInsight [7] have also focused on analyzing the performance of smartphone apps from the user's perspective. For example, by providing app developers with code-level information on the *critical path*

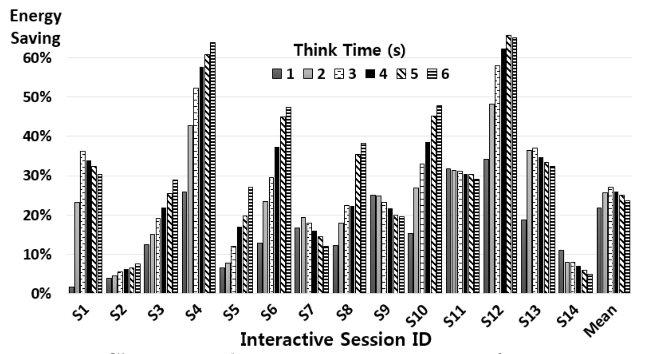


Figure 7: Changes in the average energy saving of 14 interactive sessions over varying durations of the think time.

Table 2: A comparison of normalized user-perceived response times of seven second interactive sessions over varying durations of the think time in the first interactive sessions.

Interactive Session ID	Think Time (s)					
	1	2	3	4	5	6
S2	1.00	1.00	1.01	1.01	1.00	1.00
S4	1.00	1.01	0.99	1.01	1.00	0.99
S6	1.00	1.00	1.01	1.00	1.00	1.00
S8	0.99	0.99	0.99	1.01	0.99	0.99
S10	0.98	1.02	0.96	0.99	0.99	0.99
S12	1.01	0.99	1.00	1.01	0.99	0.99
S14	0.99	0.97	0.98	1.00	1.03	0.98

for every user interaction, AppInsight can help the developers diagnose performance bottlenecks of their apps from the user's perspective. While AppInsight is effective in guiding the developer to improve user experience, our approach is different from AppInsight in that we focus on *transparent* system-level optimizations using such high-level information, instead of requiring the developer to modify the app.

Several groups have also proposed CPU frequency-scaling techniques by taking account of the quality of user experience. In particular, SmartCap [8] has shown that the neural network-based inference model can be useful to decide the minimal acceptable frequency without degrading user experience. The technique proposed in AURA [9] is also similar to our approach. In their approach, AURA can effectively decide the CPU frequency for each interactive session by exploiting an app classification scheme based on the user interaction intensity. They propose CPU frequency setting algorithms based on the Markov Decision Process using the app classification scheme. However, our work is fundamentally different from existing CPU frequency scaling techniques in that we take advantage of the user-perceived response time as a main hint for adjusting the CPU frequency.

Our proposed technique can be viewed as a simplified version of vertically-integrated OS-directed power management techniques such as Application Modes [10]. In Application Modes, for example, an app provides several working modes with different data fidelity while consuming different amount of power. Using a narrow interface between the OS and apps, during run time, the OS informs the app of mode transitions by which the limited battery capacity can be better managed. The OS makes such a transition taking account of vertically-collected information such as user preference, available application modes and the remaining battery capacity. By interpreting as a mode transition event the detection of the end of I^{perc} in the Dalvik VM, which signifies an important fidelity change point from the user's perspective, our approach can be viewed as a simplified version of Application Modes. However, our proposed approach does not require any effort from app developers, which we believe the main advantage of our approach over Application Modes.

6. CONCLUSIONS

We have presented **ura**, a user-perceived response time analyzer for Android-based smartphones, and a new CPU power management framework based on **ura**. By taking advantage of the on-line identification of the user-perceived response time from **ura**, our proposed CPU power management framework allows more aggressive low-power techniques to be applied to smartphones. In order to demonstrate the effectiveness of our proposed framework, we have developed the `oninterval` *cpufreq* governor. Based on understanding and analyzing the user-perceived response time, our proposed `oninterval` governor could make aggressive DVS decisions without any negative effect on user experience. Our experimental results show that the `oninterval` governor can save the CPU power consumption by up to 65.6% over the Linux default `ondemand` governor. Our current **ura**-based power optimization framework can be further extended to manage the energy consumption of different system components such as memory subsystem and network interfaces by exploiting the user-perceived response time.

ACKNOWLEDGMENTS We would like to thank Eyal de Lara, our shepherd, and anonymous reviewers for valuable feedback that greatly contributed to improving the final version of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (MSIP) (No. 2010-0020724) and was also supported by Basic Science Research Program through NRF funded by MSIP (NRF-2013R1A2A2A01068260). The ICT at Seoul National University and IDEC provided research facilities for this study.

7. REFERENCES

[1] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the ACM International Conference on Mobile Systems*, 2010.

[2] X. Bao, M. Gowda, R. Mahajan, and R. R. Choudhury. The case for psychological computing. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications*, 2013.

[3] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *Proceedings of the ACM International Conference on World Wide Web*, 2012.

[4] V. Pallipadi, and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, vol. 2, 2004.

[5] Android Open Source Project. Monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`, 2013.

[6] X. Chen, Y. Chen, Z. Ma, and F. C. A. Fernandes. How is energy consumed in smartphone display applications? In *Proceedings of the International Workshop on Mobile Computing Systems and Applications*, 2013.

[7] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the USENIX Conference on Operating Systems Design and Implemenation*, 2012.

[8] X. Li, G. Yan, Y. Han, and X. Li. SmartCap: User experience-oriented power adaptation for smartphone's application processor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.

[9] B. K. Donohoo, C. Ohlsen, and S. Pasricha. Aura: An application and user interaction aware middleware framework for energy optimization in mobile devices. In *Proceedings of the IEEE International Conference on Computer Design*, 2011.

[10] M. Martins, and R. Fonseca. Application modes: A narrow interface for end-user power management in mobile devices. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications*, 2013.