A Resource-Driven DVFS Scheme for Smart Handheld Devices

YU-MING CHANG, National Taiwan University
PI-CHENG HSIU, YUAN-HAO CHANG, and CHE-WEI CHANG, Academia Sinica

Reducing the energy consumption of the emerging genre of smart handheld devices while simultaneously maintaining mobile applications and services is a major challenge. This work is inspired by an observation on the resource usage patterns of mobile applications. In contrast to existing DVFS scheduling algorithms and history-based prediction techniques, we propose a resource-driven DVFS scheme in which resource state machines are designed to model the resource usage patterns in an online fashion to guide DVFS. We have implemented the proposed scheme on Android smartphones and conducted experiments based on real-world applications. The results are very encouraging and demonstrate the efficacy of the proposed scheme.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Design, Experimentation, Management, Measurement, Performance

Additional Key Words and Phrases: Energy savings, dynamic voltage/frequency scaling, resource-driven state machines, smart handheld devices, Android

ACM Reference Format:
Chang, Y.-M., Hsiu, P.-C., Chang, Y.-H., and Chang, C.-W. 2013. A resource-driven DVFS scheme for smart handheld devices. ACM Trans. Embedd. Comput. Syst. 13, 3, Article 53 (December 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2539036.2539049

1. INTRODUCTION

The tremendous advances in commodity semiconductor and communication technologies in recent years have led to a dramatic growth in the market for smart handheld devices, especially Android phones and tablets.[1] To meet the growing demand for more creative, exciting applications and services, next-generation smart handheld devices will provide significant improvements over current devices in terms of functionality and performance; yet, ironically, the trend will lead to a substantial increase in energy consumption and raise a major challenge in sustaining mobile applications and services. Recent studies of user activity indicated that efforts to improve energy efficiency should focus on the screen and CPU of mobile devices [Shye et al. 2009, 2010].

[1]comScore Inc., `http://www.comScore.com`; Strategy Analytics, `http://www.strategyanalytics.com`.

This work was supported in part by the National Science Council under grants. 101-2219-E-002-002 and 100-2628-E-001-003-MY2.

Authors' addresses: Y.-M. Chang, Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan; email: r98944014@csie.ntu.edu.tw; P.-C. Hsiu (corresponding author), Research Center for Information Technology Innovation (CITI), and Institute of Information Science (IIS), Academia Sinica, Taipei 115, Taiwan; email: pchsiu@citi.sinica.edu.tw; Y.-H. Chang, Institute of Information Science, Academia Sinica, Taipei 115, Taiwan; email: johnson@iis.sinica.edu.tw; C.-W. Chang, Research Center for Information Technology Innovation, Academia Sinica, Taipei 115, Taiwan; email: chewei@citi.sinica.edu.tw.

DOI: http://dx.doi.org/10.1145/2539036.2539049

We observe that (1) the execution of different applications may require diverse combinations of hardware resources; and (2) the usage patterns of the resources are dependent on the applications. In particular, for different applications, the usage of most resources is linked to the usage of the CPU in some way. This observation inspired us to explore the resource usage patterns of mobile applications and design a *resource-driven DVFS scheme* for smart handheld devices.

Dynamic voltage frequency scaling (DVFS) is a key technique that reduces the energy dissipation of the CPU by adjusting the supply voltage and operating frequency dynamically as the workload varies. In a pioneer paper, Yao et al. [1995] proposed a scheduling model for reducing CPU energy which spawned extensive studies on DVFS. Since then, many excellent DVFS scheduling algorithms have been proposed for real-time applications. The objective is to ensure that such applications can be executed on real-time systems without violating their deadline requirements, while minimizing the energy consumption [Aydin et al. 2001; Chen et al. 2005; Ishihara and Yasuura 1998; Mejia-Alvarez et al. 2004]. In particular, for periodic real-time applications, Aydin et al. [2001] proposed an offline algorithm for minimizing energy consumption with different power characteristics. For aperiodic real-time applications, Yao et al. [1995] proposed an offline optimal algorithm along with an online algorithm with a competitive ratio. For a CPU with discrete operating frequencies, Chen et al. [2005] developed an approximation algorithm that determines an operating frequency for each application task. These algorithms provide rigid theoretic frameworks for real-time DVFS scheduling when the execution behavior of applications, such as the worst-case execution time [Mohan et al. 2010], is highly predicable or known a priori.

Thanks to the efforts of researchers and system developers, DVFS-enabled CPUs have become a major stream in the semiconductor market, and they have been gradually incorporated into most smart handheld devices. In the past decade, various DVFS schemes have been explored for different mobile applications. One of the applications attracting lots of attention is multimedia playing, where the decoding of each video frame needs to be finished in time (according to the frame rate) to ensure smooth multimedia playing. For energy-efficient video decoding, accurate workload prediction is the core issue in order to scale the CPU frequency appropriately, because the decoding time varies greatly depending on the frame size. To estimate the decoding time, researchers have proposed a number of effective techniques which could be classified into frame-based scaling [Pouwelse et al. 2001; Choi et al. 2002] or group-based scaling [Son et al. 2001], or both [Hamers and Eeckhout 2012]. Nurvitadhi et al. [2003] have provided a comparative study of DVFS techniques developed especially for video decoding. Design issues of DVFS schemes become more challenging in light of the interaction nature between users and handheld devices. Yan et al. [2005] introduced the concept of *user-perceived latency driven voltage scaling* for interactive applications. For 3D graphics games, Mochocki et al. [2006] proposed a *signature-based estimation technique* that adjusts the operating frequency by predicting 3D graphics workloads; while Gu and Chakraborty proposed a *control-theoretic scheme* based on the feedback from recent prediction errors [2008b] and integrated the scheme with a *frame structure-based scheme* to further improve the prediction accuracy by exploiting the frame structure of graphics games [Gu and Chakraborty 2008a]. Basically, these techniques target specific applications and lead to significant energy savings by leveraging some special characteristics of the applications.

In recent years, there has been a dramatic shift in the market for personal computing. This trend has driven the development of a growing number of mobile applications, ranging from video streams and graphics games to social interaction with friends via location-based services. Mobile applications have become more pervasive and diverse. However, there has been comparatively little research on DVFS schemes targeted towards various mobile applications on smart handheld devices. A closely related approach is *AutoDVS* [Gurun and Krintz 2005], which distinguishes common, coarse-grain application behavior and utilizes forecasting techniques to predict future behavior and guide voltage scaling. The *Governor* [Pallipadi and Starikovskiy 2006], which was introduced in Linux 2.6 as the default DVFS scheme and has been adopted by Android since version 1.5, adjusts the operating frequency in each sample period according to various policies with respect to CPU utilization. Most existing approaches use only the history of CPU utilization as the basis for prediction.

In this article, we propose a resource-driven DVFS scheme for reducing the CPU's energy consumption when executing various mobile applications on smart handheld devices. The rationale behind the scheme is that it explores the resource usage patterns of mobile applications in an online fashion to guide DVFS. Specifically, we design a mechanism, called a *resource state machine,* to represent and maintain the resource usage patterns of a mobile application. In a state machine, each state is associated with a correlation table that indicates the interplay between the CPU usage and that of other hardware resources. Based on the mechanism, we propose a frequency-scaling policy to efficiently predict the CPU's subsequent workload and adjust the operating frequency accordingly. Our primary objective is to improve the energy efficiency without sacrificing the execution quality of mobile applications. Moreover, our DVFS scheme is especially applicable to long-lived applications, such as watching YouTube in relaxing time, listening to MP3 music during a long commute, looking for nearby locations on Google Map, among others. To validate the practicability of the proposed scheme, we incorporated it into Android[2] without modifying existing mobile applications and discussed the technical issues that arose from the implementation. Finally, based on real-world case studies on smartphones of HTC Desire HD,[3] we conducted experiments to evaluate the efficacy of the scheme and derive further insights into resource-driven DVFS for smart handheld devices.

The remainder of this article is organized as follows. In Section 2, we consider mobile applications and their resource usage patterns. In Section 3, we present the design concepts of the proposed resource-driven DVFS scheme. The experimental results are reported in Section 4. Section 5 contains some concluding remarks.

2. MOBILE APPLICATIONS AND THEIR RESOURCE USAGE PATTERNS

The foreground application usually dominates the user's attention. In addition, a mobile application normally needs multiple hardware resources, such as the CPU and the mobile network, to execute a task, and different applications might have different resource usage patterns. Some readers might point out that usage patterns among resources are not irrelevant to their applications. In particular, the usage of most resources is linked to the CPU usage in some way for different applications. As shown in Figure 1, the resources used by YouTube in video streaming demonstrate the relationship between the resource usage of the CPU, the mobile network, and the RAM when a buffer is implemented with the RAM to store the video data received from the Internet via the mobile network. When a certain amount of data is stored in the buffer, the video decoder is invoked to decode the video frames, which involves CPU-intensive computation. The decoded frames are then put in another buffer and displayed in succession on the screen at a constant frame rate. Thus, in video streaming applications, apart from the computational demands on the CPU, the mobile network and the LCD are busy when receiving network packets and displaying video frames, respectively. The RAM is also accessed frequently.

[2]`http://www.android.com.`

[3]`http://www.htc.com.`

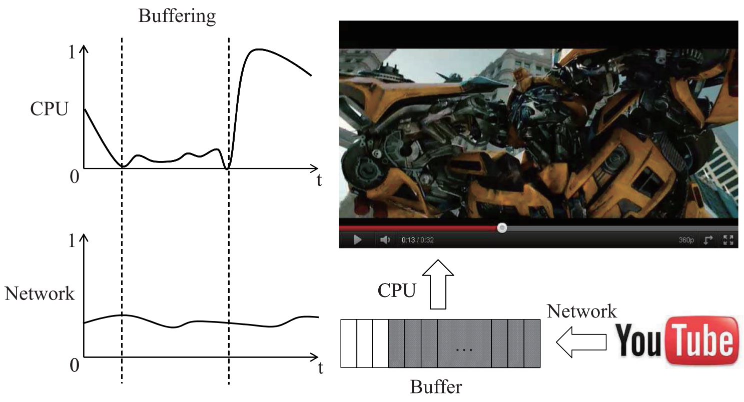
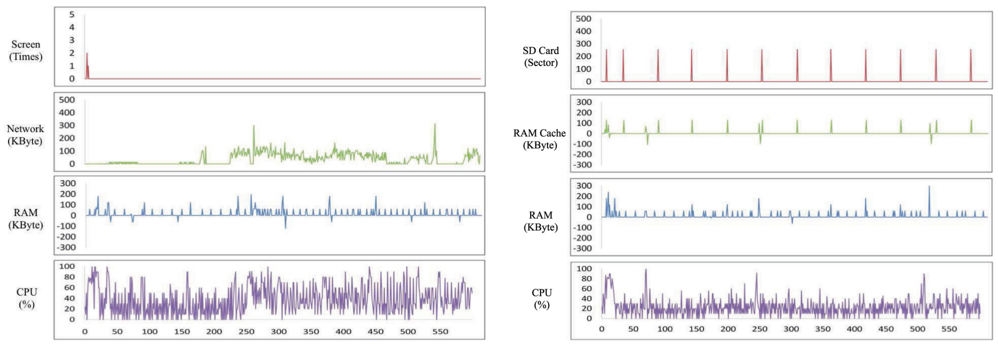


Fig. 1. The hardware resources used by video applications.



(a) YouTube. (b) MP3 music.

Fig. 2. Resource usage patterns.

To better understand the resource usage patterns of mobile applications, especially the interplay between the CPU and other hardware resources, we developed a lightweight logging mechanism to log how mobile applications use the resources of a smart handheld device. Figure 2(a) shows the traces collected from an Android smartphone of HTC Desire HD when we browsed video streams on YouTube. The traces show that the transmission rate of the mobile network varies significantly over time, and that determines the buffer's filling speed and the instant workload for video decoding. This has a positive impact on the variations in CPU utilization and the frequency of memory-access activities. In general, CPU utilization increases with the transmission rate of the mobile network and leads to access of the memory. We also investigated the resource usage patterns of playing MP3 music on the HTC Android smartphone in Figure 2(b). Interestingly, we found that accessing music files forms regular usage patterns on the SD card, and the interplay between the resources used by the MP3 application is different to that of the YouTube application. This indicates that a mobile application may have specific resource usage patterns. Moreover, the usage of each resource, especially the CPU, may be linked to the usage of other resources.

Based on our investigations, the resource usage patterns of a mobile application can be represented by a state machine. Figure 3 shows a state machine that represents the resource usage patterns of a mobile application when a user watches YouTube videos. The traces were collected by our lightweight logging mechanism. In this state machine,

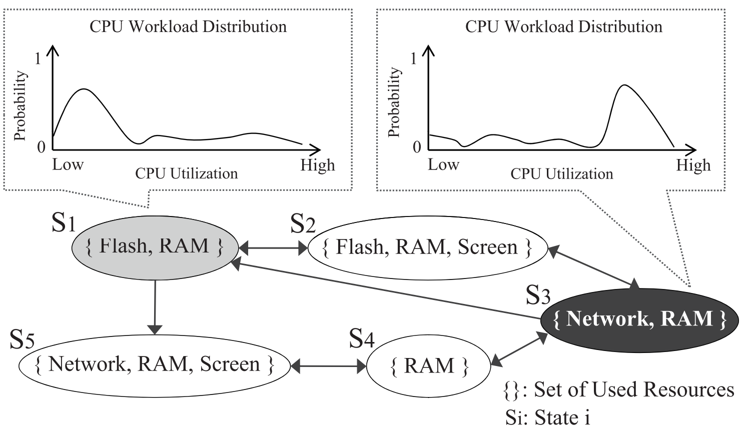


Fig. 3. A state machine for a video application.

each state represents a combination of the hardware resources that are in use, and an application is said to be staying in a state when it is using the combination of the hardware resources represented by the state. When an application stays in a state, it has a tendency to transit to some specific states and to utilize the CPU in a special way. For example, when the application stays in State S_3, it uses the mobile network and the RAM, but it might move to S_1, S_2, or S_4. The CPU utilization is usually high when the application uses the mobile network and the RAM (State S_3), and low when it uses the flash memory and RAM (State S_1). This demonstrates that a state machine can maintain the relations between different resource usage patterns. However, there is a technical problem in modeling the patterns efficiently and maintaining the information about the interplay between the CPU and other hardware resources. To address the issue, we should design a DVFS scheme that can predict the CPU workload and adjust the CPU operating frequency accurately. Then, we will be able to improve the energy efficiency of smart handheld devices without sacrificing the execution quality of mobile applications.

3. A RESOURCE-DRIVEN DVFS SCHEME

3.1. Overview

In this section, we propose a resource-driven (RD) DVFS scheme (referred to as the RD-DVFS scheme) to reduce the energy consumption of smart handheld devices. The scheme dynamically adjusts the CPU frequency based on the resource access pattern of each mobile application. To predict and adjust the CPU frequency accurately, a *resource monitor* and a set of *resource state machines* are designed to derive the interplay between the CPU utilization and the resource usage of the application. In each sampling period, the resource monitor periodically samples the usage of multiple resources to correlate the CPU utilization with the resource usage. The resource state machine is a per-process graph that maintains the resource usage patterns (i.e., the workflow of the patterns) of a mobile application. When an application becomes a foreground one, the corresponding resource state machine is activated to maintain its resource usage patterns. It is also updated with the new resource usage and the new CPU utilization patterns sampled by the resource monitor. With the updated resource state machine, the RD-DVFS scheme can identify the resource usage patterns and the interplay between the CPU utilization and the resource usage in order to predict and adjust the CPU frequency accurately.

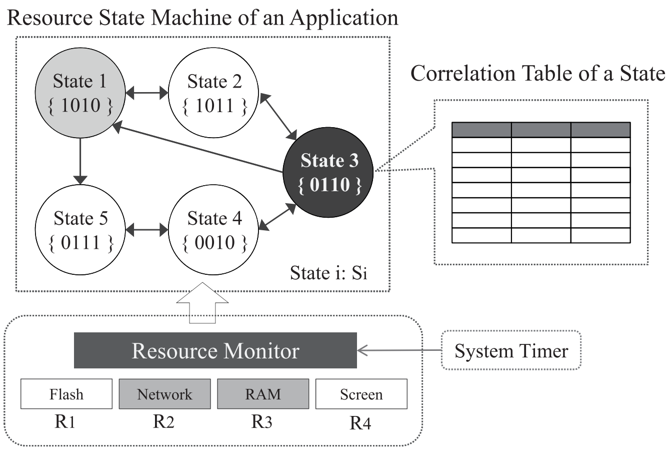


Fig. 4. An example of the RD-DVFS scheme.

Figure 4 shows an example of the framework of the RD-DVFS scheme. The resource monitor samples the usage of the flash memory, network, RAM, and screen. The combined usage of the resources forms a "state" of the resource state machine for its corresponding application. When the combination of resources changes, the application moves to the state corresponding to the new combination. Before the application leaves the original state, the CPU utilization and the next state of the original state are updated and stored in the correlation table of the original state (discussed in Section 3.2). Then, the application enters the state corresponding to the new combination of resources, and the CPU frequency is predicted and adjusted according to the information maintained in the correlation table of the newly entered state (discussed in Section 3.3). Note that each state of the resource state machine includes a correlation table that maintains the information related to the state. Because the resource state machine creates states dynamically based on the new sampled input and then refines the CPU utilization and possible next states of each state in an online fashion, the integrity of the state machine for an application becomes more complete as the execution progresses. Based on a more complete state machine, we can adjust the CPU frequency more accurately to reduce the energy consumption.

3.2. A Resource State Machine and Its Correlation Tables

3.2.1. A Resource State Machine. A resource state machine is designed to keep track of the resource usage patterns of a mobile application and determine the interplay between the CPU utilization and the resource usage of the application. For ease of presentation, each resource is considered as in use (referred to as "status 1") or not in use (referred to as "status 0"). Thus, the usage of the sampled resources can be represented as a status set that is a bitmap (or a bit string) in which each bit corresponds to the status of one resource. Each status set or bitmap represents a unique "state" in the resource state machine of a mobile application. Each state also includes a correlation table to maintain the information required to predict the subsequent CPU utilization (discussed in Section 3.2.2). In other words, a state includes two fields: a bitmap, which represents the state, and a pointer, which indicates the corresponding correlation table. For example, the state machine of the YouTube application shown in Figure 3 could be represented as the resource state machine shown in Figure 4; thus, when the network and the RAM are in use, the application stays in State S_3 whose representative bitmap is "0110". The possible next states of State S_3 are S_1, S_2 and S_4.

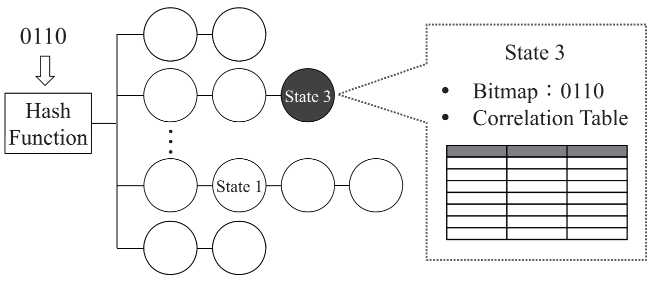


Fig. 5. A hash-based resource state machine.

Initially, a resource state machine only contains one state, which represents the status set of the resources currently used by its corresponding application. When the status set is changed, a new state is created to represent the new status set if the resource state machine does not have a state for the new status set already. Then, the state corresponding to the new status set becomes one of the next states of the current state, and the application moves to this state from the current state. Before the application leaves the current state, the CPU utilization and next states of the current state are updated accordingly. Finally, the application transits to the state corresponding to the new status set, and the CPU frequency is adjusted according to the information maintained in the correlation table of the newly entered state. It is worth noting that the number of states of a resource state machine might become large over time if the number of sampled resources is large.

As shown in Figure 5, the states of the resource state machine are indexed by hash lists. The objective is to reduce the main-memory space needed to maintain the next states of each state and to make locating the state that the application moves to more efficient. The bitmap of each state is hashed or mapped to a hash value, and states with the same hash value are added to the same hash list. When the status set of the resource usage changes, the state corresponding to the new status set can be found easily by searching the hash list indexed with the hash value of its corresponding bitmap.

3.2.2. A Correlation Table. To maintain the information related to states of the resource state machine, each state includes a correlation table comprised of three fields that keep track of, respectively, the state's next states (the N field), the CPU utilization (the U field) of the application staying in the state, and the probabilities (the P field) of the application moving to its next states. The table keeps track of the interplay between the CPU utilization and the resource usage of its corresponding state, and predicts the CPU utilization when the application enters the state that corresponds to the table. For example, Figure 6 shows the correlation table of State S_3 in a resource state machine. State S_3 has three next states (i.e., S_1, S_2, and S_4). When the application stays in S_3, it has a 50% chance of transiting to S_1 after leaving S_3, and the average CPU utilization of staying in State S_3 is 0.24 if the application moves to S_1 after leaving S_3. Note that the number of records in a correlation table is usually very small because a state normally has a limited number of next states.

Based on our observation of real mobile application traces, the P and U fields of each record in a state's correlation table can be derived from the following two properties of a resource state machine: the *affinity of state transition* (for the P field) and the *regularity of resource usage* (for the U field). The affinity of state transition means that an application had a tendency to transit to some specific state from its current state during a recent period. The regularity of resource usage means that the quantities of

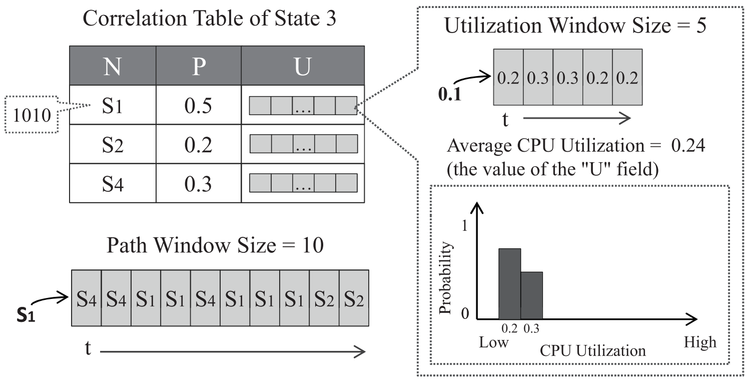

Correlation Table of State 3
N
P
U
1010
S1
0.5
S2
0.2
S4
0.3
Path Window Size = 10
S1
S4 S4 S1 S1 S4 S1 S1 S1 S2 S2
t
Utilization Window Size = 5
0.1
0.2 0.3 0.3 0.2 0.2
t
Average CPU Utilization = 0.24
(the value of the "U" field)
Probability
1
0
0.2 0.3
Low
CPU Utilization
High

Fig. 6. The correlation table before the update procedure.

resources used were similar during the recent period so that the CPU utilizations of an application staying in the same state are similar in the recent time period.

Because of the affinity of state transition, the probability of the state transition from the current state to a specific next state can be derived by maintaining a queue [Cormen et al. 2001] (called the *path window*) for each state. The path window keeps track of the destination states (i.e., the next states) of its corresponding state in several previous state transitions. Suppose the size of the path window is 10. As shown in Figure 6, the sequence of destination states of State S_3 in the previous ten state transitions from S_3 was $S_4, S_4, \ldots, S_2$ (according to the path window). Thus, the probabilities of transiting from S_3 to S_1 and S_2 were 0.5 and 0.2, respectively.

According to the regularity of resource usage, the CPU utilization in each record of a correlation table can be derived by maintaining a queue (called the *utilization window*) for each record in the correlation table. The utilization window keeps track of the CPU utilization of an application that stays in a state and will transit to the state corresponding to this utilization window. Let the size of the utilization window for each record in the correlation table equal 5. Then, as shown in Figure 6, in the previous five transitions from S_3 to S_1, the sequence of CPU utilizations of the application staying in State S_3 was 0.2, 0.3, 0.3, 0.2, and 0.2; thus, the average CPU utilization of the application was 0.24. Note that the CPU utilization of an application staying in a state is derived by monitoring the ratio of the accumulated busy time to the elapsed time of the application staying in the state. The CPU utilization should be normalized to the maximum CPU frequency in order to minimize the memory space required to store the CPU utilization record, as well as to compare the CPU utilization between different CPU frequency levels. For example, if the CPU can support an operating frequency up to 800 MHz, but it runs at 200 MHz for 40% of the busy period when an application stays in a specific state, then the application's CPU utilization will be represented as 40% utilization $\times \frac{200MHz}{800MHz} = 0.1$ normalized utilization.

When an application needs to transit to another state from its current state, the path window and utilization window of the correlation table of the current state should be updated before the application leaves the current state. With the updated path window and utilization window, the up-to-date values of the P and U fields of the correlation table could be used to help with the prediction of the CPU frequency when the application enters this state next time. After that, the application enters the next state and checks the state's correlation table to calculate the predicted CPU frequency when it stays in this state.

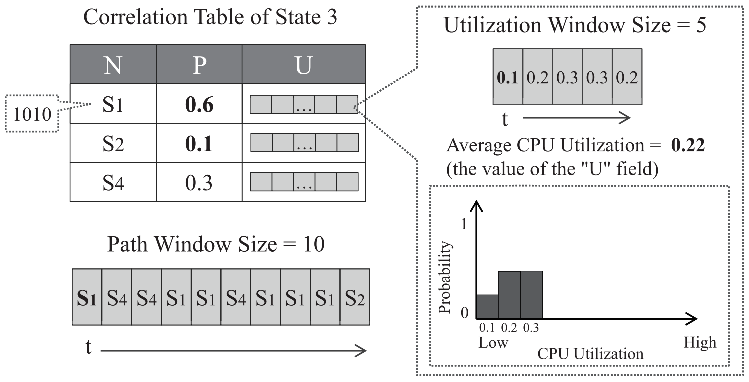

Correlation Table of State 3
N
P
U
1010
S1
0.6
S2
0.1
S4
0.3
...
Path Window Size = 10
S1 S4 S4 S1 S1 S4 S1 S1 S1 S2
t
Utilization Window Size = 5
0.1 0.2 0.3 0.3 0.2
t
Average CPU Utilization = 0.22
(the value of the "U" field)
Probability
1
0
0.1 0.2 0.3
Low
High
CPU Utilization

Fig. 7. The correlation table after the update procedure.

Figure 6 and Figure 7 show, respectively, the correlation table before and after the update procedure during the state transition. Suppose that an application initially stays in State S_3 (shown in Figure 6) and is about to transit to S_1. Before it moves to S_1, the bitmap of S_1 is added to the tail of the path window, and the bitmap of the oldest state S_2 (i.e., the state at the head of the path window) is removed because the path window is full, as shown in Figure 7. Thus, the probability of the application transiting from S_3 to S_1 is increased to 0.6 according to the information in the updated path window. Meanwhile, the value 0.1 (i.e., the CPU utilization of the application staying in this state) is added to the tail of the utilization window of the record corresponding to S_1, and the oldest CPU utilization 0.2 (i.e., the CPU frequency at the head of the corresponding utilization window) is removed because this utilization window is full. According to the updated utilization window, the average CPU utilization (i.e., the value of the U field) of the application in S_3 before it transits to S_1, becomes 0.22 (as shown in Figure 7). After updating the corresponding path window and utilization window of the correlation table of S_3, the application is ready to move to the next state (i.e., S_1) and use the correlation table of S_1 to predict the new CPU frequency. We discuss this aspect in the next section.

3.3. Frequency-Scaling Policy

To reduce the energy consumption of smart handheld devices, we propose a frequency-scaling policy that predicts and adjusts the CPU frequency by monitoring the resource usage and updating the resource state machine. Based on the policy, a procedure is activated periodically to adjust the CPU frequency. It is implemented in four phases (see Algorithm 1): the Monitor Phase (Lines 1–5), the Update Phase (Lines 6–7), the Transition Phase (Lines 8–17), and the DVFS Phase (Line 18).

Suppose there are N_r resources $\{r_1, \ldots, r_{N_r}\}$ and the CPU of a smart handheld device supports M operating frequencies $\{f_1, \ldots, f_M\}$ such that $f_1 < f_2 < \cdots < f_M$ and the current operating frequency f_c of the CPU is one of the supported operating frequencies, where $1 \leq c \leq M$. Let H denote the set of hash lists used to index the states of the resource state machine, R denote the bitmap of the current state, and s denote the current state of the application. The ratio of the accumulated busy time t_b to the elapsed time t_e of the application's current state is used to calculate the CPU utilization when the application stays in that state.

When the algorithm is invoked, it enters the Monitor State to check the status of each resource and whether the status set of the resources has changed (Lines 1–5). If

ALGORITHM 1: Frequency-Scaling Policy

```
    Input: $H$, $R$, $s$, $f_c$, $t_e$, $t_s$, $\{r_1, \ldots, r_{N_r}\}$, $\{f_1, \ldots, f_M\}$
    Output: $R$, $s$, $f_c$
 1  $R_n = 0$;                                                  /* Monitor Phase */
 2  for $i = 1$ to $n$ do
 3      if GetStatus($r_i$) == 1 then
 4          $R_n = R_n \mid (1 \ll (i\text{-}1))$;
 5  if $R == R_n$ then return ($R$, $s$, $f_c$);
 6  UpdateCT($s$, $R_n$, $\frac{t_b}{t_e} \times \frac{f_c}{f_M}$);   /* Update Phase */
 7  $R = R_n$;
 8  $p$ = SearchState($H$, $R$);                                /* Transition Phase */
 9  if $p == NULL$ then
10      $s'$ = CreateNewState($R$);
11      InsertState($H$, $s'$);
12      $s = s'$;
13      $f_c = f_M$;
14  else
15      $s = p$;
16      $u$ = GetPredictedCPUUtilization($s$);
17      $f_c = f_{l+1}, \{f_{l+1} \mid f_l \le u \times f_M < f_{l+1}\}$;
18  SetCPUFrequency($f_c$) ;                                    /* DVFS Phase */
19  return ($R$, $s$, $f_c$);
```

the status set has not changed, the procedure simply returns; otherwise, it proceeds to the Update Phase to update the path window and the utilization window of the current state's correlation table based on the bitmap R_n of the next state (Line 6), that is, to update the path window with R_n and the utilization window with $\frac{t_b}{t_e} \times \frac{f_c}{f_M}$, where $\frac{t_b}{t_e} \times \frac{f_c}{f_M}$ is the CPU utilization normalized to the maximum operating frequency f_M, because the stored CPU utilization is related to the CPU's maximum operating frequency f_M (see the example in Section 3.2.2). Then, the procedure updates the bitmap of the new status set of the resources (Line 7).

With the updated bitmap, the procedure moves to the Transition Phase and uses the updated bitmap R as the key to search the hash lists for the state corresponding to R. The objective is to derive the position of the state corresponding to R (Step 8). If such a state cannot be found, a new state is created for the resource state machine based on the updated bitmap and added to the hash lists. Then, the application moves to the new state and executes with the CPU's maximum operating frequency f_M, because we do not have any historical information related to this new state[4] (Lines 9–13). In contrast, if the state corresponding to R is found, the application moves to this state and uses the state's correlation table to predict the CPU utilization to derive the CPU frequency (Lines 14–17). Note that if the stability of the CPU frequency is considered, the predicted CPU utilization could be derived by calculating the state's expected CPU utilization, $\sum_{1 \le i \le N_s} P_i \times U_i$, where N_s is the number of records in the correlation table and P_i (resp. U_i) is the value of the P(resp. U) field in the ith record of the correlation

[4]Low CPU frequency may increase delays noticeably while switching applications, and any delays in switching could have an adverse impact on user satisfaction. This strategy is adopted to avoid response delays incurred by the quick switches between applications. However, adopting such a strategy could not achieve significant energy savings for short-lived, interaction-driven applications like appointment reminders.

table. In contrast, if the user's experience is the critical concern, the maximum CPU utilization among the records in the correlation table of the state should be selected as the predicted CPU utilization. In other words, the predicted CPU utilization could be calculated with the information maintained in the correlation table based on the user's preferences or the system's requirements. After the predicted execution frequency f_c is determined, the procedure enters the DVFS Phase to configure the CPU's execution with the new frequency f_c and then returns (Lines 18–19).

3.4. Implementation Framework and Issues

To implement the proposed RD-DVFS scheme without modifying existing mobile applications, we designed an implementation framework and integrated it into the Android smartphone of HTC Desire HD. As shown in Figure 8, the implementation framework consists of an application layer and a system layer. The latter is further divided into the user space and the kernel space. The proposed RD-DVFS scheme is implemented in the user space, so there is no need to modify the existing Android kernel or mobile applications. In Android, each mobile application runs on an individual Dalvik virtual machine in the application layer.[5] A mobile application is composed of some *activities* and *services*, where an activity presents a graphical interface that interacts with the user in the foreground, while the services run in the background to actually perform the functions. No matter how many mobile applications (one or multiple) run simultaneously, only one activity can be executed and appear in the foreground. At the same time, therefore, there will be only one foreground activity and may be multiple background services that require the CPU resource. The resource usage patterns of the foreground activity and background services are monitored by the proposed RD-DVFS scheme to ensure that the CPU is adjusted to a frequency level sufficient to handle them. Since there is only one foreground activity at a time, the monitored patterns are naturally maintained by the resource state machine of the corresponding mobile application. The RD-DVFS scheme is implemented in the user space of the system layer with two components: the resource monitor (RM) component and the resource state machine (RSM) component. The RM component is triggered by a system timer (in the kernel space) and monitors the resource usage of the running foreground application through the "Proc" file system. The RSM component constructs a resource state machine for each foreground application. Based on the resource state machine, the RSM component adjusts the CPU frequency through the DVFS library in the kernel space.

The performance of building the resource state machines is the critical issue to the success of the proposed RD-DVFS scheme and should be carefully handled in this implementation framework. Thus, the resource state machine of a mobile application should be reconstructed efficiently to predict the CPU frequency after its corresponding application has been activated (or it becomes a running foreground application). To solve this issue, a mobile application's resource state machine and its correlation tables are stored in the storage device as a file (called an RSM file); nonetheless, to reduce the memory/storage overheads, the states that are unreachable from the current state need not be saved into the RSM file, because they were created but are not recently used by their corresponding mobile application. An RSM file has a file name with a unique application ID mapped to its corresponding mobile application; thus, when a mobile application is activated as a foreground one, its corresponding resource state machine can be reconstructed immediately by loading the RSM file corresponding to the application's ID if the RSM file is not cached in the memory. We noticed that invoking or closing a background application, or switching the foreground application,

[5] `http:developer.android.com/index.html`.

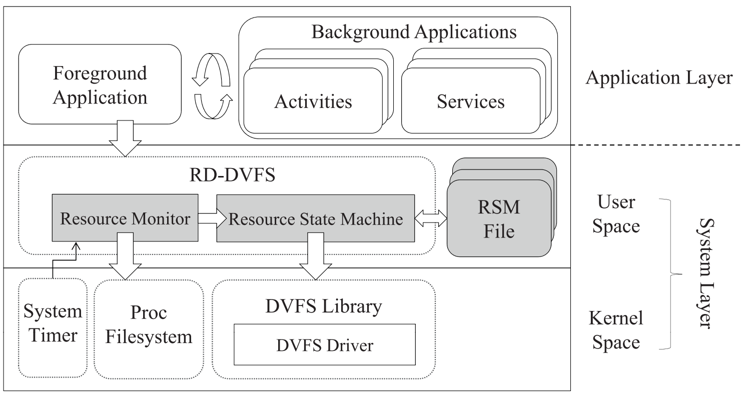


Fig. 8. The implementation framework.

could lead to a significant change in resource usage patterns. Any delays could have an adverse impact on user satisfaction. Therefore, our RD-DVFS scheme adjusts the CPU promptly to the maximum operating frequency when meeting a state that cannot be found in the resource state machine, and then starts to adapt to the new demands. We have implemented the proposed scheme on Android to conduct real-world experiments. Users were not conscious of significant delays while invoking, closing, or switching applications. This is because users' reaction times are much longer than the time required to scale the CPU to the highest frequency.

In this implementation framework, the memory overhead mainly comes from the space required to store applications' resource state machines at runtime. In a resource state machine, each state includes a 4B pointer that points to its next state, a 4B bitmap to represent the state, and a 4B index to indicate its corresponding correlation table. The number of entries in a correlation table is bounded by the path window size N_p, because the number of entries depends on the number of different state bitmaps in the path window, and an entry consists of a 4B P field, a 4B P field, and a $(4 \times N_u)$-byte U field, where N_u is the utilization window size. Thus, the size (or memory overhead) of a state is at most $12 + N_p \times (8 + 4 \times N_u)$ bytes. When $N_p = 5$ and $N_u = 5$, which are the settings used in the performance evaluation, the size of each state is no more than 152 B. In our implementation, the maximum number of states in a resource state machine is $2^{11} = 2{,}048$ since, excluding CPU, 11 types of resources are monitored. Thus, the memory overhead for a mobile application is $152\text{B} \times 2{,}048 = 304\text{KB}$ in the worst case. In practice, a resource state machine usually consists of no more than 100 states; this implies that the memory overhead for a mobile application is, on average, less than 15.2 KB. Such a memory overhead is deemed small, even when multiple applications are running simultaneously, because a smartphone usually has hundreds of megabytes of memory space. For example, the Android smartphones of HTC Desire HD used in the experiments are equipped with 768MB DRAM. On the other hand, the extra CPU overhead/utilization incurred by the proposed RD-DVFS scheme is also very small. This is because the proposed scheme (see Algorithm 1) is activated infrequently. On each activation, it only checks the resource usage and simply returns if the resource usage does not change; if the resource usage changes, only a small overhead is paid to update the corresponding state, followed by adjusting the CPU frequency when needed. Thus, the RD-DVFS scheme's resource monitor can be deemed a very lightweight daemon that does not affect the system performance significantly.

Table I. Specifications of HTC Desire HD

Hardware	
Processor	Qualcomm 8255 Snapdragon
Memory	RAM: 768MB
	ROM: 512MB
Telecommunication	HSPA/WCDMA: 900/2100 MHz
	GSM: 850/900/1800/1900 MHz
Screen	TFT-LCD 480 × 800 pixels
Network	Bluetooth 2.1
	Wi-Fi: IEEE 802.11 b/g/n
Camera	8 million pixel
Storage	SD 2.0 compatible, 4G
Battery	1340 mAh
Software	
OS	Android 2.2.1
	Linux Kernel 2.6.32

4. PERFORMANCE EVALUATION

4.1. Experimental Setup and Performance Metrics

To better understand the properties of and gain insight into resource-driven DVFS for smart handheld devices, we implemented the proposed scheme on Android to conduct experiments on real-world video streams and various real traces collected from representative mobile applications. We conducted online-based and trace-based experiments. The online-based experiments were designed to validate the practicability of the proposed scheme and compare its efficacy with the DVFS policies adopted by Android. The trace-based experiments were designed to evaluate the performance of the proposed scheme on various mobile applications and assess the extra overheads. We conducted the experiments on Android smartphones of HTC Desire HD[3], equipped with a Qualcomm 1GHz CPU, which allows operations on 18 different frequencies. The smartphone's hardware and software specifications are detailed in Table I. In the experiments, we utilized 12 types of resources: memory, memory cache, ROM read, ROM write, flash read, flash write, SD card read, SD card write, Soft interrupt, screen, network reception, and network transmission. Based on their interplay with the CPU and the frequency-scaling policy, we adjusted the smartphone's operating frequency dynamically. In the experimental environment, we used the Power Monitor of Monsoon Solutions[6] to measure the smartphone's transient power and energy consumption, as shown in Figure 9.

In the online-based experiments, we studied two videos with different characteristics, namely *Avatar* and *Monga*, both of which can be found on YouTube. Avatar, which is a typical trailer for Sci-Fi movies, comprises diverse scenes that demonstrate the movie's high fidelity. It is a high-definition video with a resolution of 720 × 576, a bit rate of 2,360 kbps, and a frame rate of 24 fps. In general, the incurred CPU workload is heavy and varies significantly. Monga is a typical action movie trailer, so the high fidelity feature is also emphasized, but the consecutive scenes are longer and more homogeneous than those of Sci-Fi movies. The video has a resolution of 480 × 270, a bit rate of 455 kbps, and a frame rate of 30 fps. Thus, the CPU workload is relatively light and static, compared to that incurred by the Avatar video. First, we investigated the impacts of the parameter settings of the proposed scheme. Specifically, we investigated the impacts of the sampling rate on the average power consumption and the average

[6]`http:www.msoon.com.`

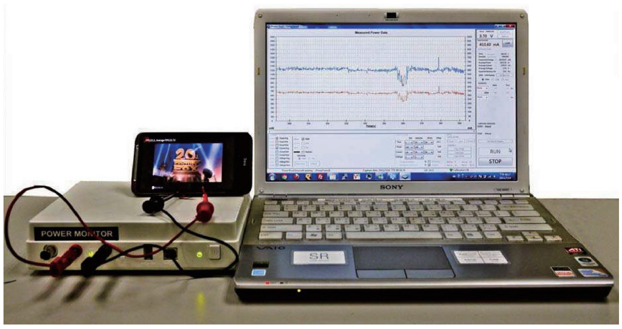


Fig. 9. The experimental environment.

frame rate when the size of each window was set at 5. We also assessed the impacts of the path (resp. utilization) window size when the utilization (resp. path) window size was set at 5 and the sample rate was set at 15 Hz.

Then we compared the RD-DVFS scheme with the Governor employed in Android under three frequency-scaling policies; *Performance*, *Ondemand*, and *Conservative*, which are all history-based approaches and use only the history of CPU utilization as the basis for prediction. The Performance policy always uses the highest CPU frequency to execute an application. The Ondemand policy adjusts the frequency to the highest level when the CPU is busy and reduces it step by step when the CPU is idle. In contrast, the Conservative policy increases the frequency step by step when the CPU is busy and drops it to the lowest level when the CPU is idle. We studied two scenarios where the videos, Avatar and Monga, were retrieved remotely via WiFi or locally from the SD Card. Moreover, we also investigated some scenarios where a video was played in the foreground while other background applications were running simultaneously. We downloaded a file from the Internet via the browser or decompressed a zip file in the background, or performed both simultaneously, while playing the Avatar video from the SD Card. With regard to the parameters used in our algorithm, the sizes of the path window and the utilization window were both set at 5, and the sample rate was set at 15 Hz. The adopted performance metric was the percentage of energy savings achieved without adversely impacting the display frame rate.

In the trace-based experiments, we studied five representative mobile applications, namely *Youtube*, *Album*, *Game*, *Map*, and *Music*. First, we assessed the extra CPU overheads incurred by the resource monitor when it sampled the resource usage of applications at different sample rates and CPU frequency levels. Then, we investigated the respective impacts of the two window sizes on the prediction accuracy. To this end, for each mobile application, we collected 50 traces via the resource monitor at a sampling rate of 20Hz. The Youtube traces were generated by playing the top 50 popular videos on YouTube. The Album, Game, and Map traces were generated by 50 people browsing a Picasa Album, playing the Homerun Battle 3D game, and searching locations on Google Map, respectively. The Music traces were generated by playing 50 MP3 songs. The adopted evaluation criterion was the prediction error, which is defined as the difference between the ground truth of the required CPU utilization and the utilization predicted when a state transition occurs. Note that, for the same application, a low prediction error implies that the proposed scheme can dynamically adjust the operating frequency accurately to achieve energy savings while ensuring that the application runs smoothly [Lee et al. 2005]. For different applications, however, a lower prediction error does not necessarily imply better performance or less energy consumption, because different applications could tolerate prediction errors

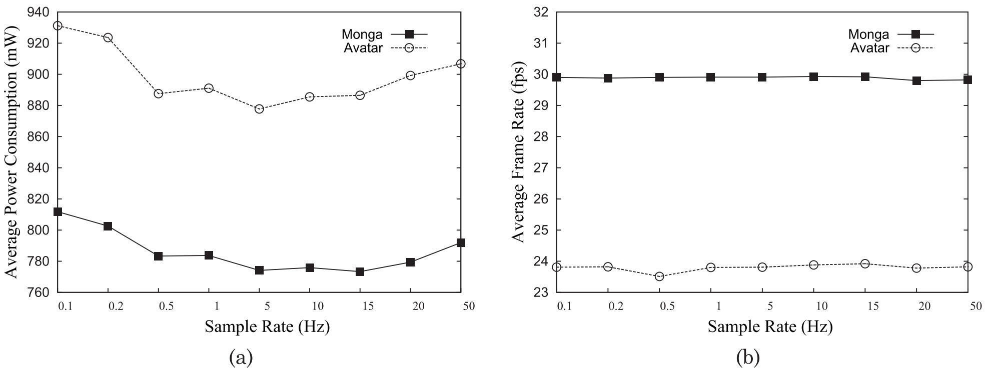


Fig. 10. Impact of the sampling rate.

with different degrees. We explored the average prediction error for each trace when both window sizes were set at 10. In addition, we assessed the impact of varying one window size on the prediction error when the other window size was set at 10. Finally, we investigated the convergence of the prediction error for various applications during the construction of the corresponding resource state machines.

4.2. Sampling Rates

Figure 10 shows the impact of the sampling rate of the resource monitor on the average power consumption and the average frame rate when the RD-DVFS scheme was applied in the online-based experiments. As shown in Figure 10(a), when the sampling rate was lower than 5Hz, the average power consumption declined for both videos as the sampling rate increased. In contrast, when the sampling rate was higher than 5Hz, the average power consumption increased gradually as the sampling rate increased. The reason for this interesting phenomenon is that a higher sampling rate makes frequency prediction more accurate (which implies more energy savings), but it also causes additional energy dissipation. Conversely, a lower sampling rate reduces the energy dissipation incurred by resource monitoring, but it delays the reaction to workload variances. The monitoring overhead, although small, could negate the energy gain if increasing the sampling rate only yields a marginal improvement in the prediction accuracy. Moreover, the Avatar video consumes more power than the Monga video because it has a higher frame rate that requires higher CPU frequencies. Importantly, the results in Figure 10(b) show that under the RD-DVFS scheme, the average frame rates approached the frame rates required for the videos, which means that the videos were displayed smoothly.

4.3. Utilization Window Sizes

Figure 11 shows the impact of the utilization window size on the average power consumption and the average frame rate when the RD-DVFS scheme was applied in the online-based experiments. From Figures 11(a) and 11(b), we observe that the utilization window size did not have a significant impact on the power consumption or the frame rate. The result evidences the regularity of resource usage, one of the two properties the proposed scheme's efficacy relies on. However, when the size was set at an extremely large value (e.g., 1,000 in Figure 11(a)), slightly more power was consumed due to a delayed reaction to workload changes. In contrast, when the size was set at an extremely small value (e.g., 1 in the figure), we observed two opposite cases. The reason for the phenomenon is that the CPU workload changes more significantly in the Avatar video than in the Monga video. Thus, switching frequencies several times

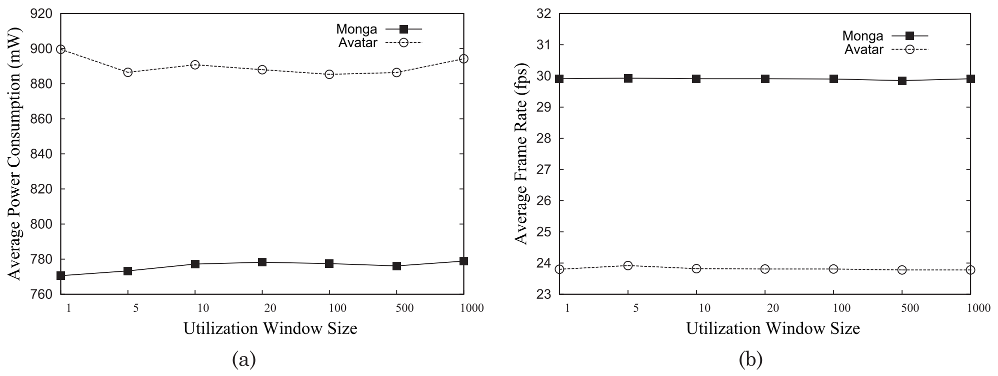


Fig. 11. Impact of the utilization window size.

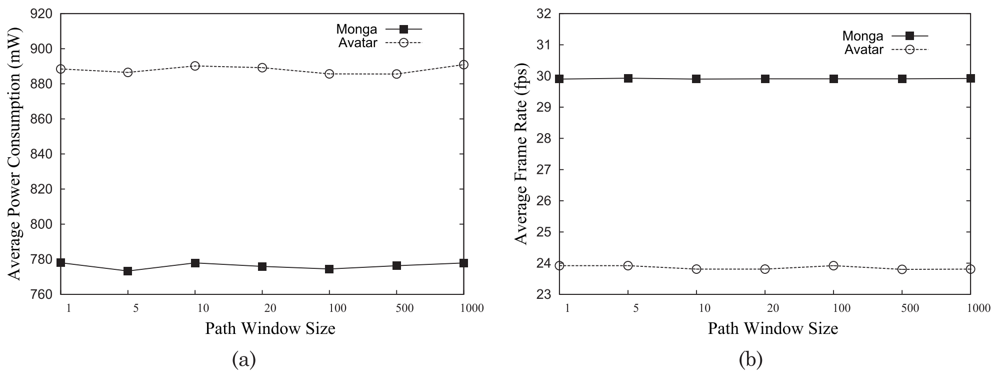


Fig. 12. Impact of the path window size.

may result in unnecessary power consumption. Overall, a small window size (but not extremely small) can reduce memory usage and yield a better performance.

4.4. Path Window Sizes

Figure 12 shows the impact of the path window size on the average power consumption and the average frame rate when the RD-DVFS scheme was applied in the online-based experiments. As shown in Figures 12(a) and 12(b), the path window size did not have a significant impact on the power consumption and the frame rate. The result is similar to that in the previous experiment, except that an extremely large window size could also catch on the tendency of state transition as reasonable window sizes. The result evidences the affinity of the state transition, the other important property that the proposed scheme's efficacy relies on. The experiment result also suggests that a small path window can reduce memory usage, now that a good performance can be achieved regardless of the window size adopted.

4.5. Performance Comparison—Video Applications

Figure 13 shows the energy savings and the average frame rates under the RD-DVFS scheme and under the Governor (with the Performance, Conservative, and Ondemand policies). In the experiment, a video, that is, Avatar or Monga, was played and retrieved remotely via WiFi or locally from the SD Card. As shown in Figure 13(a), the RD-DVFS scheme outperformed the Governor in terms of energy efficiency in all cases. We observe

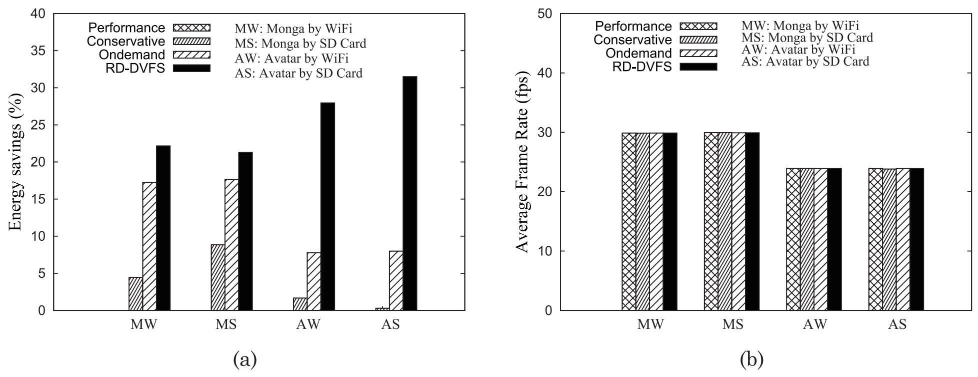


Fig. 13. Performance comparison of the RD-DVFS scheme and the Governor for video applications.

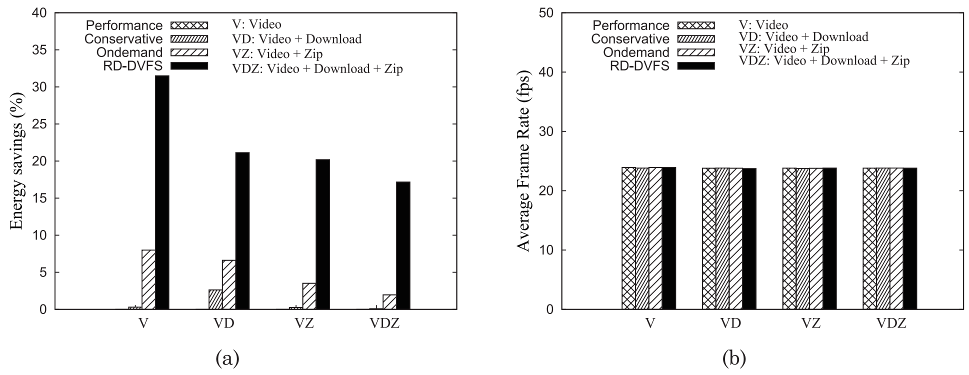


Fig. 14. Performance comparison of the RD-DVFS scheme and the Governor under four different settings with Multiple Application.

that the RD-DVFS scheme achieved more energy savings on Avatar than on Monga, whereas the Governor achieved more savings on Monga. The reason was that the Avatar video consumed more power than the Monga video when the highest CPU frequency was used, and the difference in energy savings was more evident for the Avatar video than for the Monga video under the RD-DVFS scheme. This implies that the proposed scheme can aggressively and accurately scale down the CPU frequency, especially when the CPU workload is heavy and varies significantly. In contrast, the Governor (with the Ondemand policy) performed well when the workload was relatively light, but it could not determine the appropriate times to scale down the CPU frequency for heavy workloads. In addition to energy savings, all the compared schemes can achieve the frame rates required for the videos under all settings, as shown in Figure 13(b). These results demonstrate that the proposed scheme can achieve CPU energy savings of 22–31% without adversely impacting the smoothness of video playing.

4.6. Performance Comparison—Multiple Applications

Figure 14 shows the energy savings and the frame rates achieved by the RD-DVFS scheme and the Governor (with the three policies) when some background applications were executed simultaneously. In the experiment, we report the impacts of the four possible combinations of two different background applications, namely Download and Zip, on the performance when playing the Avatar video from the SD Card. As shown in Figure 14(a), smaller energy savings were achieved under all the DVFS schemes

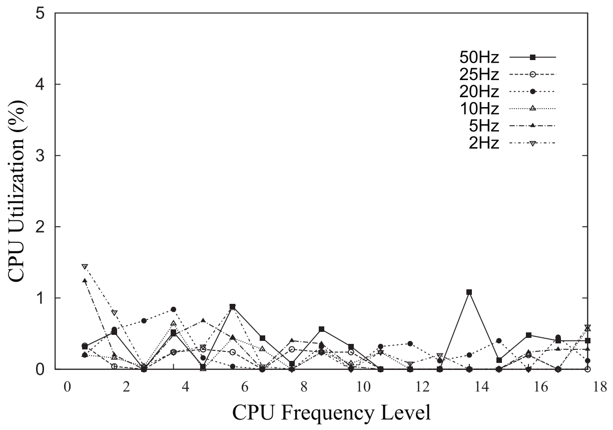


Fig. 15. Overheads under different sampling rates and frequency levels.

when more applications were executed together. This is as expected because there were fewer opportunities for the CPU to scale down the frequency when the workload was heavier. All the DVFS schemes performed worse when Zip ran in the background than when Download ran in the background, because Zip was more CPU-intensive than Download. Nevertheless, we observed an encouraging result that the RD-DVFS scheme outperformed the Governor with any policy greatly in all cases. The outperformance in energy savings was more manifest when applications of diverse characteristics were executed simultaneously. This also agrees with the aforementioned observation that the RD-DVFS scheme can aggressively and accurately scale down the CPU frequency when the workload was heavy but varied significantly, while the Governor cannot. Despite a greater challenge in energy savings with multiple applications, the proposed RD-DVFS scheme can still achieve energy savings of 17–21%, much higher than the energy savings achieved by the Governor. On the other hand, both the RD-DVFS scheme and the Governor can achieve the frame rate required for the video, as shown in Figure 14(b), and did not affect the smoothness of video playing.

4.7. Extra Overheads

Figure 15 shows the extra CPU overheads incurred by the RD-DVFS scheme in the trace-based experiments with various sampling rates at different CPU frequency levels. The x-axis and y-axis denote, respectively, the CPU frequency level and the extra CPU utilization incurred by the scheme. As shown in the figure, most of the extra CPU utilization incurred by the proposed scheme was less than 1% when the sampling rate was ranged from 2Hz to 50Hz and the CPU frequency was set at one of the 18 levels. This is because the proposed scheme (see Algorithm 1) is activated infrequently and introduces limited overheads on each activation. In theory, the CPU overhead should increase with the sampling rate, but the overhead shown in this figure is irregular. The reason for this non-intuitive phenomenon is that the RD-DVFS only incurs small overheads that are within a margin of error. Based on the results, the RD-DVFS scheme's resource monitor can be deemed a very lightweight daemon that does not affect the system performance significantly.

4.8. Prediction Errors

Figure 16 shows the average prediction error when the RD-DVFS scheme was applied to each of the 50 traces collected from the five representative mobile applications in the trace-based experiments. Table II lists the mean and standard deviation of the

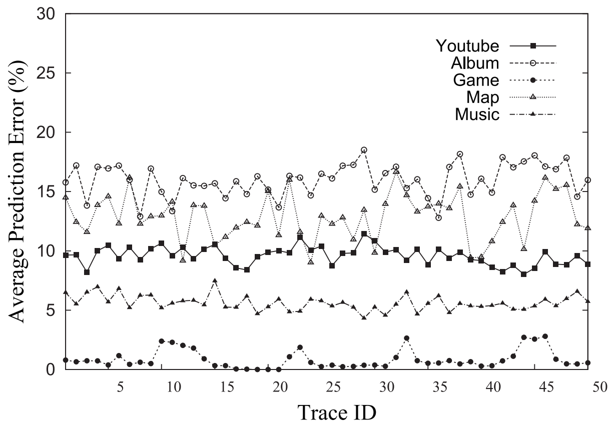


Fig. 16. Prediction errors for different types of mobile applications.

Table II. Means and Standard Deviations of the Prediction Errors for Different Applications

Application	Mean	Standard Deviation
YouTube	7.272	1.405
Album	15.985	1.380
Game	0.841	0.778
Map	12.888	1.943
Music	5.619	0.645

prediction errors of the 50 traces of each application. The results show that the proposed scheme can achieve reasonable prediction errors between 0.841% and 15.985%, depending on the characteristics of applications. In general, the prediction error is larger for interactive applications (Album and Map) than for non-interactive applications (YouTube and Music). This is reasonable because the execution of interactive applications involves user behavior. Fortunately, interactive applications usually can tolerate higher prediction errors, because users' reaction times are much longer than the time required to scale the CPU frequency. Moreover, we found an interesting exception with the Game application whose prediction error was extremely small. The reason being that the CPU workload remained high and only varied slightly during the application's execution. As a result, the application's demand for CPU utilization could be predicted accurately, but less energy saving could be achieved in order to ensure that the application runs smoothly. On the other hand, the standard deviations of the prediction errors for all the applications were quite small, ranging from only 0.645 to 1.943. This implies that the proposed scheme can catch on the resource usage patterns so that it can produce similar results for the same application.

4.9. Two Window Sizes

Figure 17 shows the impact of the two window sizes on the prediction error of the RD-DVFS scheme for different mobile applications in the trace-based experiments. In general, the average prediction error began to converge when the size was larger than 5, as shown in both subfigures. In other words, neither window size had a significant impact on the prediction error. The results evidence that the execution of mobile applications has two important properties: the regularity of resource usage and the affinity of state transitions. Importantly, the proposed scheme can exploit the two properties to appropriately perform DVFS under different applications. In addition, the results

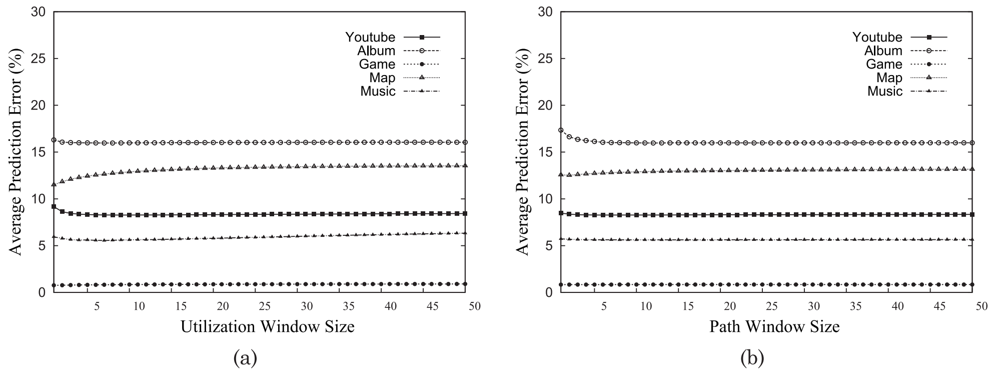


Fig. 17. Impact of the window size on the prediction error.

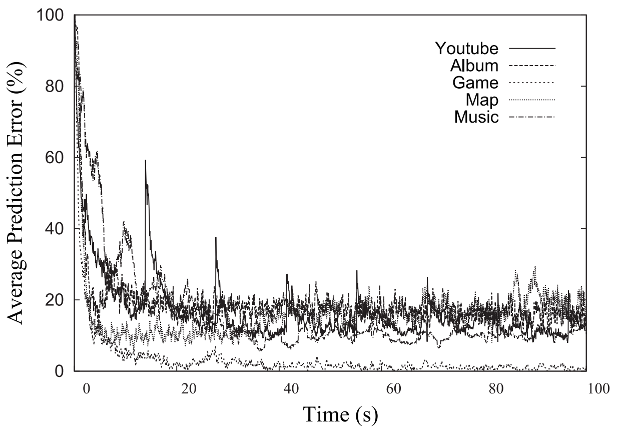


Fig. 18. Convergence time required for the prediction error.

of this experiment confirm the results presented in Figures 11 and 12, that is, a small window size is recommended for the RD-DVFS scheme.

4.10. Convergence Time

Figure 18 shows the time required for the prediction error to converge the first time the corresponding resource state machines were constructed for the five investigated applications. The x-axis and y-axis denote the convergence time and the average prediction error, respectively. As shown in the figure, the prediction error declined over time for all of the applications. This is as expected because, during the self-tuning process, a resource state machine should become more complete, and its correlation tables should become more accurate. The results show that the prediction error achieved convergence after about five to ten seconds for all the applications. The convergence time is short in practice because the execution time of an application is usually longer and the corresponding resource state machine only needs to be constructed once. We also observed that some hikes in the prediction error occurred abruptly. This was because the proposed scheme always adjusted the CPU frequency to the highest level when a new state (i.e., a new combination of used resources) appeared. However, the prediction error achieved convergence very quickly, as shown in the figure. This finding implies that the RD-DVFS scheme can adapt efficiently to resource usage patterns that may change over time because of user interaction or other factors.

5. CONCLUDING REMARKS AND FUTURE WORK

In this article, we propose a resource-driven DVFS scheme meant for various mobile applications on smart handheld devices. Based on the resource state machines developed to model the usage patterns of mobile applications on hardware resources, we present a frequency-scaling policy that adjusts the operating frequency dynamically to achieve energy savings. The scheme exploits two important properties of the resource usage patterns of mobile applications: the affinity of state transitions and the regularity of resource usage. To evaluate the scheme, we implemented it on Android[2] and conducted experiments on real-world applications. Under the proposed scheme, smartphones of HTC Desire HD[3] achieved CPU energy savings of 22–31% when playing videos without adversely impacting the smoothness of video playing. Meanwhile, the experiments based on real-world traces provided useful insights into resource-driven DVFS for mobile applications on smart handheld devices.

In our future research, we will investigate a multicore architecture, which is one of the major design focuses now to meet the emergence of computation-intensive applications that run on smart handheld devices. We will also consider how to reduce the energy dissipation of the CPU for short-lived, interaction-driven applications which may play a large role in some users' living scenarios.

REFERENCES

AYDIN, H., MELHEM, R., MOSSÉ, D., AND MEJÍA-ALVAREZ, P. 2001. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 225–232.

CHEN, J.-J., KUO, T.-W., AND SHIH, C.-S. 2005. $1 + \epsilon$ approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor. In *Proceedings of the IEEE/ACM International Conference on Embedded Software (EMSOFT)*. 247–250.

CHOI, K., DANTU, K., CHEN, W.-C., AND PEDRAM, M. 2002. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 732–737.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. The MIT Press, Cambridge, MA.

GU, Y. AND CHAKRABORTY, S. 2008a. A hybrid DVS scheme for interactive 3D games. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 3–12.

GU, Y. AND CHAKRABORTY, S. 2008b. Control theory-based DVS for interactive 3D games. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 740–745.

GURUN, S. AND KRINTZ, C. 2005. AutoDVS: An automatic, general-purpose, dynamic clock scheduling system for hand-held devices. In *Proceedings of the IEEE/ACM International Conference on Embedded Software (EMSOFT)*. 218–226.

HAMERS, J. AND EECKHOUT, L. 2012. Exploiting media stream similarity for energy-efficient decoding and resource prediction. *ACM Trans. Embed. Comput. Syst. 11,* 1, 2:1–2:25.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 197–202.

LEE, B., NURVITADHI, E., DIXIT, R., YU, C., AND KIM, M. 2005. Dynamic voltage scaling techniques for power efficient video decoding. *J. Syst. Architect. 5,* 10–11, 633–652.

MEJIA-ALVAREZ, P., LEVNER, E., AND MOSSÉ, D. 2004. Adaptive scheduling server for power-aware real-time tasks. *ACM Trans. Embed. Comput. Syst. 3,* 2, 284–306.

MOCHOCKI, B. C., LAHIRI, K., CADAMBI, S., AND HU, X. S. 2006. Signature-based workload estimation for mobile 3D graphics. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 592–597.

MOHAN, S., MUELLER, F., ROOT, M., HAWKINS, W., HEALY, C., WHALLEY, D., AND VIVANCOS, E. 2010. Parametric timing analysis and its application to dynamic voltage scaling. *ACM Trans. Embed. Comput. Syst. 10,* 2, 25:1–25:34.

NURVITADHI, E., LEE, B., YU, C., AND KIM, M. 2003. A comparative study of dynamic voltage scaling techniques for low-power video decoding. In *Proceedings of the European Symposium on Algorithms (ESA)*. 292–298.

PALLIPADI, V. AND STARIKOVSKIY, A. 2006. The ondemand governor: Past, present and future. In *Proceedings of the Linux Symposium*. Vol. 2. 223–238.

POUWELSE, J., LANGENDOEN, K., LAGENDIJK, R., AND SIPS, H. 2001. Power-aware video decoding. In *Proceedings of the Picture Coding Symposium (PCS)*. 303–306.

SHYE, A., SCHOLBROCK, B., AND MEMIK, G. 2009. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the IEEE/ACM MICRO*. 168–178.

SHYE, A., SCHOLBROCK, B., MEMIK, G., AND DINDA, P. A. 2010. Characterizing and modeling user activity on smartphones: Summary. In *Proceedings of the ACM SIGMETRICS*. 375–376.

SON, D., YU, C., AND KIM, H. 2001. Dynamic voltage scaling on MPEG decoding. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 633–640.

YAN, L., ZHONG, L., AND JHA, N. K. 2005. User-perceived latency driven voltage scaling for interactive applications. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 624–627.

YAO, F., DEMERS, A., AND SHENKER, S. 1995. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*. 374–382.

Received October 2011; revised August 2012; accepted October 2012