User-Centric Scheduling and Governing on Mobile Devices with big.LITTLE Processors

PI-CHENG HSIU, Academia Sinica
PO-HSIEN TSENG, WEI-MING CHEN, and CHIN-CHIANG PAN, National Taiwan University
TEI-WEI KUO, National Taiwan University and Academia Sinica

Mobile applications will become progressively more complicated and diverse. Heterogeneous computing architectures like big.LITTLE are a hardware solution that allows mobile devices to combine computing performance and energy efficiency. However, software solutions that conform to the paradigm of conventional fair scheduling and governing are not applicable to mobile systems, thereby degrading user experience or reducing energy efficiency. In this paper, we exploit the concept of *application sensitivity*, which reflects the user's attention on each application, and devise a user-centric scheduler and governor that allocate computing resources to applications according to their sensitivity. Furthermore, we integrate our design into the Android operating system. The results of experiments conducted on a commercial big.LITTLE smartphone with real-world mobile apps demonstrate that the proposed design can achieve significant energy efficiency gains while improving the quality of user experience.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-Time Systems and Embedded Systems*

General Terms: Algorithms, Design, Experimentation, Human Factors

Additional Key Words and Phrases: Scheduling, DPM, DVFS, user experience, energy efficiency, big.LITTLE, mobile systems

1. INTRODUCTION

The tremendous paradigm shift in personal computing has led to an explosive growth in the number of mobile apps on Google Play[1]. To support more complicated mobile applications, mobile devices will increasingly improve the computing performance, such as employing more cores and/or higher frequencies. However, the performance gain usually results in higher power consumption, which is obviously a significant concern

[1]Google Play, http://play.google.com.

A preliminary version [Tseng et al. 2014] of this paper appears in the Proceedings of the IEEE/ACM Design Automation Conference (DAC) 2014.
This work was supported in part by the Ministry of Science and Technology of the Republic of China under grant Nos. 102-2221-E-001-007-MY2 and 100-2221-E-002-120-MY3, and by the "The Core Technologies of Smart Handheld Devices (1/4)" of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China.
Authors' Addresses: P.-C. Hsiu, Research Center for Information Technology Innovation (CITI), and Institute of Information Science (IIS), Academia Sinica, Taipei 115, Taiwan; email: pchsiu@citi.sinica.edu.tw; P.-H. Tseng, W.-M. Chen, C.-C. Pan, Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan; email: r00922073@csie.ntu.edu.tw, r02922032@csie.ntu.edu.tw, d98922036@csie.ntu.edu.tw; T.-W. Kuo, Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, and Research Center for Information Technology Innovation, Academia Sinica, Taipei 115, Taiwan; email: ktw@csie.ntu.edu.tw.

DOI: http://dx.doi.org/10.1145/0000000.0000000

for mobile devices. As mobile applications will become more diverse as well, heterogeneous computing architectures, especially ARM's big.LITTLE[2], are deemed a promising solution for the emerging genre of mobile devices. The intention behind big.LITTLE is to create a computing unit that combine the performance of a power-hungry processor with the energy efficiency of a low-speed processor. Some recent mobile devices, such as Samsung Galaxy S4[3], have featured the latest big.LITTLE architecture.

Hardware solutions have been sought to satisfy the two conflicting requirements, namely performance and energy efficiency, simultaneously. Unfortunately, software solutions that borrow legacy designs directly from some conventional operating systems, e.g., Linux, cannot be applied seamlessly in mobile systems. In the paradigm of conventional *fair scheduling*, the *scheduler* treats all applications *equally* by default, and it allocates the available cores and execution cycles to running applications in an equal manner. When an application causes an abrupt increase in the CPU workload, if the *governor* blindly scales up the CPU frequency and/or turns on more cores to meet the application's needs, energy efficiency may be degraded with no improvement in user experience. In contrast, if the governor does not react to the workload changes or even reduces the available computing resources to save power, the response time of the application that dominates the user's attention may be affected by the other applications and damage the user's experience. This is because the governor treats applications equally. Therefore, the scheduler and governor developed for mobile devices should be different from previous designs for personal computers and servers.

Mobile applications provide a large variety of functionalities, some of which are delay-sensitive while others are delay-tolerant. Delay-tolerant applications, like file zipping, can be delayed without affecting the user's experience significantly. By contrast, delay-sensitive applications, such as video playing, are extremely sensitive to delay and thus require timely responses. Because different applications' delay may have different impacts on user experience, mobile applications should be treated *unequally* by allocating computing resources to them *proportionally* according to "some metric" that reflects user perception. *Proportional-share scheduling*, in which tasks receive computing resources proportional to their weights or priorities, is the norm in many operating systems [Li et al. 2009] and real-time systems [Chandra et al. 2001]. Any proportional-share scheduling algorithm could be implemented by a *multilevel feedback queue scheduler*, the most general CPU-scheduling framework that can be configured to match a specific system under consideration [Silberschatz et al. 2010]. However, configuring an appropriate scheduler for a specific system requires some means by which to assign priorities and allocate resources to tasks.

In the past decades, a massive body of research on *priority assignment* and *resource allocation* has accumulated against real-time systems. Classical real-time scheduling algorithms assign priorities to tasks according to their *deadlines* or *periods*, and always allocate the CPU to the highest-priority task [Liu and Layland 1973]. For *soft* real-time systems, where tasks do not have explicit deadlines, *resource reservation* was proposed as an alternative to classical scheduling algorithms [Abeni and Buttazzo 1998]. Reservation-based algorithms differ from each other in how resources are preallocated to tasks. In [Abeni and Buttazzo 1999], for example, each task is characterized by an *importance value* specified by the user, and the resources will be shared according to tasks' importance. The concept of resource reservation was implemented as an extension in a number of operating systems, such as Linux [Cucinotta et al. 2004] and Android [Segovia et al. 2010], to add real-time behavior and guarantee computing resources to tasks, without jeopardizing the systems [Maia et al. 2010]. By contrast,

[2]ARM, http://www.arm.com
[3]Samsung, Inc., http://www.samsung.com.

mobile systems emphasize particularly on user experience. In [Mercati et al. 2013], applications are classified into highly critical or less critical, and a highly-critical application is always executed with the highest CPU frequency. The idea can be realized with a friendly interface allowing the user to mark applications with different *critical levels* in Android [Mercati et al. 2014]. To favor the application visible on the screen, the latest versions of Android impose *control groups* on all applications to ensure that the foreground application receives a lot more CPU time than background applications [Goransson 2014]. However, a simultaneous improvement in user experience and energy efficiency cannot be achieved by simply allocating different amounts of CPU time to foreground and background applications.

In this paper, we introduce the concept of *application sensitivity* into scheduler and governor designs for mobile systems, with the objective of improving user experience and energy efficiency simultaneously. The sensitivity of each application is to reflect the degree of attention it receives from the user, so that more computing resources (not only temporal but also "spatial" CPU resources) can be allocated to applications that attract more user attention at the moment. Realizing this concept in mobile operating systems obviously raises several design challenges. First, determining the sensitivity of each application is a major challenge. Our design, which is based on some observations about human attention and interaction [Chang et al. 2013; Tolia et al. 2006], classifies the sensitivity of applications into three levels: *high* (or interactive), *medium* (or foreground and system), and *low* (or background). Based on these levels, we define three *sensitivity states*, as well as delineate the rules of *sensitivity inheritance* and *transitions* between the states. Note that we determine an application's sensitivity based on its current status, instead of its category. Then, another challenge is how to exploit each application's sensitivity during scheduling and governing. This is difficult because the governor tends to limit the computing resources (i.e., the used cluster, the number of active cores, and the operating frequency of the CPU) to prevent unnecessary power usage. Consequently, the scheduler has to allocate the available computing resources appropriately to maintain the quality of user experience. Our design, which considers user experience and energy efficiency simultaneously, manages and allocates computing resources in certain proportions to applications with high, medium, and low sensitivity. Specifically, the scheduler exploits the sensitivity of applications to perform *thread prioritization*, *thread allocation*, and *thread migration*; and the governor considers the sensitivity when managing computing resources with *dynamic power management* (DPM), *dynamic voltage and frequency scaling* (DVFS), and *cluster switching*. Third, to validate the concept and evaluate our design, we have integrated our user-centric scheduler and governor into the Android operating system. We discuss some technical issues that arise with the integration. Finally, we conducted extensive experiments on a commercial Samsung Galaxy S4 smartphone with some mobile apps found on Google Play. Compared to the *Completely Fair Scheduler* [Love 2010] and the *On-demand governor* [Pallipadi and Starikovskiy 2006], the proposed design can reduce the CPU's energy consumption by 24.9% while improving 35.2% response time of the application that dominates the user's attention. The experiment results also provide some valuable insights into user-centric scheduling and governing on mobile devices.

The remainder of this paper is organized as follows. Section 2 provides some background information and an example to show the potential benefits. In Section 3, we discuss the design details and implementation issues. The experiment results are reported in Section 4; and Section 5 provides a review of related work. Section 6 contains some concluding remarks.

2. BACKGROUND AND MOTIVATION

First, we provide some background information about the scheduler and governor in mobile operating systems. Then, we present an example to demonstrate the potential benefits of user-centric scheduling and governing over traditional fair scheduling and governing, and consider the major design challenges.

2.1. Background Information

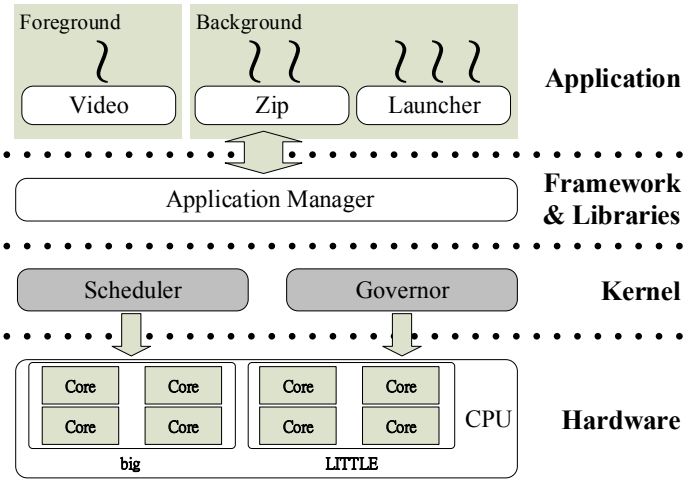


Fig. 1. The system architecture.

Figure. 1 shows the architecture of the mobile system considered in this work. The application layer allows multiple applications to be executed simultaneously; however, only one can run in the foreground at a time, while the others run in the background[4]. The *application manager* in the framework layer maintains the metadata of the running applications; for example, details of the application currently running in the foreground. The kernel layer contains two key components, namely, the scheduler and the governor. The scheduler supervises thread execution, such as selecting the next thread to be executed, the duration of the execution, and the core to be used. Therefore, it has a significant influence on each thread's performance. Meanwhile, the governor manages the CPU's resources by scaling its frequency up or down, turning the CPU cores on or off (i.e., switching to a low-power state), and switching the CPU to the LITTLE or big cluster according to changes in the workload. Although vendors may employ different scheduling and governing implementations, a general approach is based on CPU utilization as follows.

The scheduler handles thread prioritization, thread allocation, and thread migration. The Completely Fair Scheduler (CFS) [Love 2010] is widely used in modern operating systems, including Android, so we utilize it as an example here. The key concept of the CFS is to allocate available computing resources *fairly* to threads based on their priorities and to balance the utilization among different cores in order to achieve ideal multitasking.

[4]This is to faithfully reflect the current practice of mobile operating systems, but our design can easily be extended without the limitation.

(1) Thread Prioritization. Thread prioritization assigns a specific priority to each thread and then allocates the CPU time per *scheduling period* to running threads based on their priorities. In the CFS, the default priority of a thread is inherited from its parent, and all threads forked by user applications have the same priority. If two threads with the same priority are running on the same core, they will be allocated the same amount of CPU time per scheduling period. The CFS maintains the running threads in a red-black tree. In each scheduling period, every running thread will be executed exactly once, and the thread that has spent the least amount of time in the tree will be executed first to achieve fairness. Note that the available CPU time is reallocated when the number of running threads changes, i.e., when threads join or leave the running queue. In other words, an active core will not be idle unless there are no running threads.

(2) Thread Allocation. Thread allocation assigns a newly forked thread to an active core for execution. When a new thread is forked, the CFS checks the number of running threads on each core and allocates the new thread to the core with the smallest number of running threads.

(3) Thread Migration. Thread migration moves threads between cores to prevent a workload unbalance among the cores. The CFS periodically checks if the number of running threads on one core is more than twice the number on another core. In each *sampling period*, if thread migration is triggered, a thread on the core with the largest number of running threads is moved to the core with the smallest number of running threads. This process is repeated until the workload is balanced.

The governor introduced in Linux 2.6 is still used by Android to adjust the CPU frequency in order to save energy. In this paper, it represents the component responsible for managing computing resources via DPM, DVFS, and cluster switching.

(1) DPM. DPM attempts to prevent unnecessary power usage by turning cores on or off. A popular way to decide whether to turn cores on or off is to check the operating frequency per sampling period. If the current frequency exceeds a predefined threshold, another core is turned on immediately to provide more computing resources. Conversely, if the frequency is lower than another threshold for a few sampling periods, one active core is turned off to save power. Moreover, some variant implementations may confine the number of active cores to the number of running threads. Note that turning cores on or off is usually accompanied by thread migration.

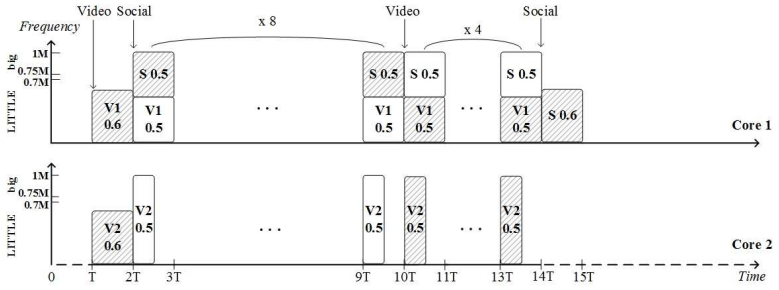
(2) DVFS. DVFS adjusts the operating frequency (and thus the supply voltage) dynamically as the CPU utilization varies. Recently developed mobile devices only support synchronous scaling, i.e., all the active cores have to operate on the same frequency level in each sampling period. A number of scaling polices have been proposed. Among them, *Ondemand* [Pallipadi and Starikovskiy 2006] is a default policy implemented on many commercial mobile devices. When a core's utilization exceeds a predefined threshold, Ondemand scales the operating frequency to the highest level to deal with bursty workloads. In contrast, when the utilization of all cores falls below another predefined threshold, it scales down the frequency step by step to save power. Note that different clusters should have appropriate pairs of thresholds.

(3) Cluster Switching. Cluster switching reconfigures the CPU to accommodate dynamic computing needs. Although there are three models for the CPU to be rearranged, only the simplest model, i.e., one cluster is active at a time, is implemented in the latest mobile devices. With the simplest implementation, the big cluster is activated if a core's utilization exceeds a predefined threshold while the LITTLE cluster already operates on its highest frequency. Similarly, the transition to the

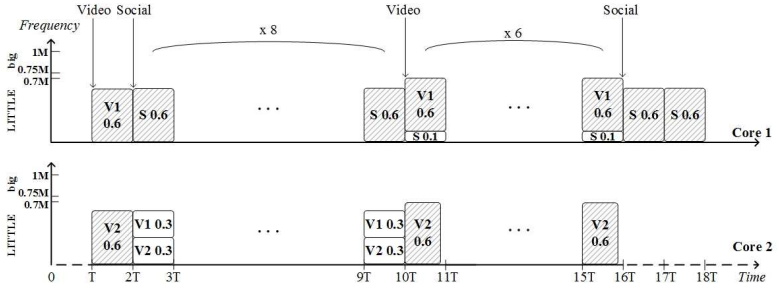
LITTLE cluster occurs when the utilization of all cores falls below another threshold while the big cluster operates on its lowest frequency. To alleviate the ping-pong effect, the two thresholds should be set carefully with a sufficiently large difference between them. Moreover, immediately before a cluster is powered off, the other will be activated with the same number of cores, so that thread migration can be performed on a one to one basis.

2.2. Example of Benefits

Consider a scenario where two applications, a social app and a video app, run on a mobile device equipped with two different dual-core clusters, as shown in Figure 2. Each core of the LITTLE cluster operating at the highest frequency level can provide 0.7 mega computing cycles per scheduling period T. For the big cluster, each core can provide 0.75 to 1 mega computing cycles, depending on the operating frequency. The social app has one thread that requires 6.6 mega computing cycles to complete its execution[5]. Moreover, when executed in the foreground, it requires 0.6 mega cycles per period to achieve smooth user interaction. The video app comprises two synchronized threads, each of which also requires a total of 6.6 mega computing cycles as well as 0.6 mega cycles per period to ensure smooth video playing in the foreground. The video app is started at time T; then, the social app is started at time $2T$, so the video app is switched to the background. At time $10T$, the video app is switched back to the foreground again. When the video is finished, the social app is switched back to the foreground and executed until completion.



(a) Fair scheduling and governing



(b) User-centric scheduling and governing

Fig. 2. A motivating example.

[5]To make the example simple, we assume that an application requires the same number of computing cycles no matter which cluster it runs on, although the big can often execute more instructions in a cycle than the LITTLE. Moreover, the overhead required for thread migration is ignored in the example.

Figure 2(a) shows the result derived based on the scheduler and governor described in Section 2.1. Initially, the device is idle with the LITTLE cluster being active, and core 2 is in the off mode to save power. When the video app starts at time T, core 2 is turned on and the two video threads, denoted by V_1 and V_2, are allocated individually to the cores so that each thread can obtain 0.6 mega cycles during the scheduling period. The social app starts running at time $2T$. As both cores are occupied by one thread, the social thread, denoted by S, is allocated randomly to core 1. Because all application threads are treated equally by the scheduler, the social thread has to share the available computing cycles evenly with the video thread. To achieve smooth user interaction, the governor activates the big cluster and attempts to increase the CPU frequency. However, the social app is only allocated 0.5 mega cycles per period, although the cores are already operating at the highest frequency level. As a result, the user may not interact with the social app smoothly until the video app is switched back to the foreground at time $10T$. Similarly, because each of the two synchronized video thread is only allocated 0.5 mega cycles per period, the user may not experience smooth video playback until the video finishes at time $14T$. At that time, the LITTLE cluster is also activated with core 2 turned off to save power. Finally, the social app is executed in the foreground on core 1 until completion at time $15T$. Treating all threads equally may, ironically, affect the user's experience.

The foreground application usually dominates the user's attention. In user-centric scheduling and governing, delay-sensitive and delay-tolerant threads are allocated different numbers of computing cycles to improve the user's experience and save energy simultaneously. We use the example in Figure 2(b) to explain the potential benefits. Suppose the computing resource allocated to a foreground thread is six times that allocated to a background thread on the same core. When the video app starts at time T, the scheduler treats its two threads equally as usual. At time $2T$, the social app starts to run in the foreground. To achieve smooth user interaction, the scheduler moves V_1 to core 2 and allocates core 1 exclusively to S; then, it reallocates the computing cycles to the three threads based on their sensitivity. Consequently, the social app is executed with sufficient computing cycles per period until the video app is switched back to the foreground at time $10T$. Similarly, to achieve smooth video playback, the scheduler moves V_1 back to core 1 and reallocates the computing cycles accordingly. After the video finishes playing at time $16T$, the social app is switched back to the foreground. Because the social thread only needs 0.6 mega cycles per period, the governor turns off core 2 and scales down the CPU frequency to save power. Finally, the social app finishes at time $18T$. Although the two apps finish later than in Figure 2(a), the user may not be aware of the delay, and may even feel satisfied due to smooth user interaction and video playing. Moreover, this schedule requires less energy than the previous schedule derived in Figure 2(a) because it uses a frequency lower than or, at least, equal to that used in the previous schedule to deal with the same part of the CPU workload. Note that the social app can also stay in the background to save more energy. In other words, user-centric scheduling and governing is designed to react to user behavior to improve user experience and save energy.

2.3. Design Challenges

Although the previous example demonstrates the benefits of user-centric scheduling and governing, several design challenges must be resolved in order to realize the concept, in addition to the issue of compatibility with existing mobile operating systems.

Determining Thread Sensitivity: The first challenge is how to determine the sensitivity of each application to reflect the user's perception of its delay. A straightforward way is to categorize mobile applications based on how they affect user experience and assign them fixed sensitivity; however, such a way cannot react to individual user

behavior. This is particularly difficult because an application could have different degrees of sensitivity for individual users depending on how they use it. Even for the same user, an application's sensitivity might vary over time as the focus of the user's attention changes.

Scheduling and Governing Based on Sensitivity: Another challenge is how to perform scheduling and governing based on thread sensitivity so that both user experience and energy efficiency can be improved. Intuitively, the scheduler should allocate more computing resources to threads with higher sensitivity than those with lower sensitivity. However, computing resources are finite because the governor simultaneously attempts to limit the available resources to save power. Therefore, the scheduler and governor should cooperate to find a balance between user experience and energy efficiency for threads with different levels of sensitivity.

3. USER-CENTRIC SCHEDULING AND GOVERNING

In this section, we explain how thread sensitivity is determined (Section 3.1); present the design details of our user-centric scheduler and governor (Section 3.2); and discuss some technical issues that arise when integrating our design into the Android operating system (Section 3.3).

3.1. Thread Sensitivity Determination

3.1.1. Observations. First, we make two observations about human attention and interaction that provide useful insights into determining thread sensitivity.

Foreground Domination: Mobile applications with a large number of creative functionalities induce various user habits and behavior patterns [Falaki et al. 2010]. However, a thread's sensitivity is highly dependent on the application that the user is currently focusing on. Because the foreground application usually dominates the user's attention [Chang et al. 2013], this observation suggests that threads belonging to the foreground application should be differentiated from those belonging to background applications.

Highly Sensitive Interaction: Most mobile users interact with their devices frequently and sometimes switch between mobile applications quickly. Studies of human computer interaction indicate that people usually expect to receive feedback from their devices within a few hundred milliseconds after each touch interaction [Donohoo et al. 2011; Tolia et al. 2006]. To avoid an adverse impact on user experience, if the user is interacting with the foreground application, we should provide as many computing resources as possible to ensure a timely response.

3.1.2. Sensitivity States and Transitions. Based on the above observations, we classify thread sensitivity into three levels: high (interactive), medium (foreground and system), and low (background). However, there are different ways to classify sensitivity. For example, we can further divide the background applications into two sensitivity levels, depending on whether they use the audio interface. In contrast, a simple distinction between foreground and background applications could also be useful. In the following, we discuss the three sensitivity states, as well as possible transitions of threads between the states.

Sensitivity States: Figure 3 shows the transitions between the high, medium, and low sensitivity states. Once an application thread is forked, it will remain in one of the three states until its termination. All threads belonging to an application should be in the same state because they may be interdependent. Thus, all the threads belonging to the current foreground application are deemed medium or high sensitivity threads, depending on whether the user is interacting with the application. In contrast, all threads belonging to background applications are classified as low sensitivity threads

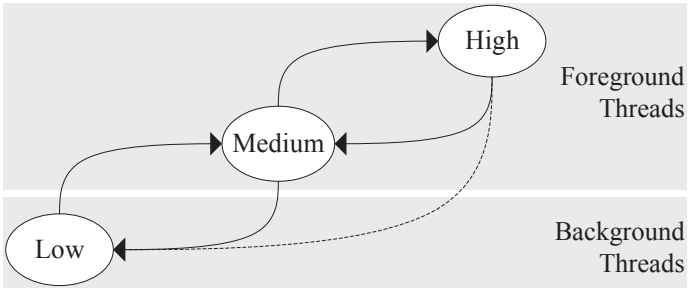


Fig. 3. Sensitivity states and thread transitions.

because they attract less user attention. Moreover, in our design, any system threads that do not belong to user applications are categorized as medium sensitivity threads and their sensitivity will never change.

Sensitivity Inheritance and Transitions: The initial sensitivity of application threads is derived from the root parent of all user applications. For example, the first thread of any application in Android is created by a system service called *Zygote*, whose sensitivity is set as medium. When an application is started in the foreground, the first thread inherits its sensitivity from Zygote, and subsequent threads inherit their parents' sensitivity. Next, we describe the possible transitions of application threads.

Medium $\rightarrow$ *Low*. This transition occurs when the foreground application is switched to the background. When a switch takes place, all threads belonging to the foreground application will change from the medium to the low sensitivity state.

Low $\rightarrow$ *Medium*. This transition occurs when a background application is switched to the foreground. Note that there is always a foreground application. In current practices, if the foreground application is terminated, or switched to the background, it will be replaced by the previous application or the home user interface. All the threads of the new foreground application will transit from the low to the medium state.

Medium $\rightarrow$ *High*. This transition occurs when a user interacts with a mobile device. When the user touches the screen or any button, all application threads whose sensitivity is medium will transit to the high sensitivity state because only one application can run in the foreground.

High $\rightarrow$ *Medium*. This transition occurs when the user does not interact with the mobile device for a few hundred milliseconds[6] after an interaction. In other words, a thread will only remain in the high state for a short time after each touch because the response to the touch is supposed to be finished in a few hundred milliseconds. All the threads with high sensitivity will revert to the medium state.

Display Toggle: There is also a special case that is caused by the display toggle. When the display is turned off, all application threads are forced into the background and therefore change to low sensitivity. As soon as the display is turned on, all the threads revert to their original states, i.e., their states before the display was turned off.

3.2. Scheduler and Governor Designs

[6]Following the measurements reported in [Tolia et al. 2006], our implementation adopts 500 ms.

3.2.1. Design Principles. Before explaining how our design exploits the sensitivity states, we present some principles that our scheduler and governor follow to find a balance between the two conflicting objectives discussed earlier.

User Experience Considerations: To improve user experience, more computing resources should be allocated to threads with higher sensitivity than those with lower sensitivity. Thus, our scheduler allocates the CPU time per scheduling period according to whether the threads are in the high, medium, or low sensitivity state. It also tends to distribute high- and medium-sensitivity threads to active cores by balancing the total sensitivity of the threads on those cores.

Energy Efficiency Considerations: To reduce energy consumption, the management of computing resources should be based on the sensitivity of the running threads. To this end, our governor allocates as many resources as possible to high-sensitivity threads, sufficient resources to medium-sensitivity threads, and limited resources to low-sensitivity threads. Moreover, given the same number of threads and the same number of cores, it has been observed that imbalanced utilization will consume more energy than balanced utilization [Wei et al. 2010]. Thus, our scheduler attempts to balance not only the sensitivity but also the utilization among cores.

3.2.2. Scheduler Designs. Next, we explain how our scheduler exploits thread sensitivity to perform thread prioritization, thread allocation, and thread migration.

Thread Prioritization: The scheduler allocates the CPU time per scheduling period to running threads based on their sensitivity. For this operation, our implementation borrows the Android *priority system*, in which the priority of a thread depends on a value, called the *nice value*, in the range -20 to 19. A smaller value implies a higher priority. If the difference between the nice values of two threads is k, the CPU time allocated to the higher-priority thread is approximately 1.25^k times that allocated to the lower-priority thread; thus, the larger the value of k, the more unequally[7] the threads are treated. After the running threads are prioritized, the scheduler relies on the CFS to schedule them for execution. Note that thread prioritization is performed whenever a thread's sensitivity changes.

Thread Allocation: Each newly forked thread is allocated by the scheduler to an active core based on its sensitivity. Once forked, the thread inherits its parent's sensitivity. If the sensitivity is medium or high, the thread is allocated to the core with the minimum *total sensitivity* so that it can obtain more computing resources. In contrast, if its sensitivity is low, it is allocated to the core with the minimum *total workload* so as to balance the utilization of the cores. The total sensitivity of a core is the sum of the sensitivity values[8] of the threads running on it. Similarly, the total workload is the sum of the computing cycles of the running threads. We explain how to estimate the workload of a running thread later in the discussion of implementation issues in Section 3.3.

Thread Migration: The scheduler attempts to balance the cores' sensitivity as well as their utilization. This is achieved by periodically triggering thread migration. In each sampling period[9], the first step involves moving threads from the core with the

[7]Our implementation maps high, medium, and low sensitivity states to the nice values of -20, -10, and 10, respectively. The settings, which are adjustable, are to demonstrate that the low-sensitivity threads can be treated extremely unequally without affecting user experience significantly.

[8]Our implementation sets the sensitivity values of high-, medium-, and low-sensitivity threads at 3, 2, and 1 respectively. These settings are sufficient to distribute high- and medium-sensitivity threads to cores.

[9]Thread migration has a non-negligible overhead. Based on our measurements, it requires approximately 0.06 ms to move one thread from one core to another and 1.6 ms for all threads to migrate from one cluster to the other on Samsung Galaxy S4. In our implementation, the sampling period is set at 200 ms to allow a trade-off between the extra overheads and the reaction time to workload changes.

maximum total sensitivity to the core with the minimum total sensitivity. Threads are moved one by one sequentially until immediately before the total sensitivity of the source core becomes lower than that of the destination core. Moreover, it is better to move threads with higher sensitivity, and the tie is broken at random if multiple threads have the same sensitivity. The first step, which attempts to balance the sensitivity, allows higher sensitivity threads to run on a core with fewer threads because the core with the minimum sensitivity usually has fewer threads. The next step balances the utilization by moving threads one by one sequentially from the core with the maximum total workload to the core with the minimum workload, until immediately before the total workload of the source core becomes smaller than that of the destination core. In this step, threads with lower sensitivity are preferred. They are moved in FIFO order so that the threads moved in the first step will not be moved back to the original core.

3.2.3. Governor Designs. We now explain how the governor considers the sensitivity when managing computing resources with DPM, DVFS, and cluster switching.

DPM: For each sampling period, based on the total workload and the CPU power model, the governor periodically turns cores on or off to prevent unnecessary power usage. In a multi-core system, turning off as many cores as possible will not necessarily save power because the power consumed by a core is a convex increasing function of the operating frequency [Mutapcic et al. 2009]. We use Samsung Galaxy S4 that adopts the big.LITTLE architecture as an example. Based on our measurements, for the LITTLE cluster (with an ARM Cortex-A7 quad-core processor), using two cores is more effective when using one core has to operate on a frequency over 450 MHz, provided that the total workload can be evenly distributed between the two cores. Moreover, using three cores is more beneficial when using two cores has to operate on a frequency over 450 MHz, and using four cores is more beneficial when using three cores has to operate on a frequency over 500 MHz. For each sampling period of 200 ms, one, two, and three cores operating on 450, 450, and 500 MHz can provide $450\times0.2=90$, $450\times0.2\times2=180$, and $500\times0.2\times3=300$ mega computing cycles respectively. Similarly, the three thresholds for the big cluster (with an ARM Cortex-A15 quad-core processor) are 900, 1000, and 1300 MHz, which can provide 180, 400, and 780 mega computing cycles respectively. In other words, for each cluster, the number of cores to be turned on simply depends on whether the total workload in a sampling period exceeds the three thresholds. However, the total workload yielded in the preceding period highly depends on the sensitivity of the running threads (more details will be discussed later in the DVFS design). In addition, the number of active cores obviously should not be larger than the number of running threads. Note that if an active core is turned off, the threads running on the core must be moved to other active cores and treated as if they are newly forked threads, as mentioned previously in the discussion of thread allocation.

DVFS: After the number of active cores has been determined, the governor selects an appropriate operating frequency for this sampling period according to the total workload and the sensitivity of the threads running on each core. Specifically, if any threads are in the high sensitivity state, the highest frequency level is selected directly so as to provide them with as many computing resources as possible. For each interaction, because the threads only remain in that state for a very short time, energy will not be wasted unnecessarily and the user's experience will be improved significantly. Otherwise, if there are no high-sensitivity threads, the operating frequency is selected as follows. Let us consider any of the active cores. For medium-sensitivity threads on the core, the computing cycles they need will be completely satisfied. In contrast, the computing cycles of low-sensitivity threads may only be partially satisfied. Our implemen-

tation treats low-sensitivity threads as unequally as possible and only provides them with the *least* computing resources, given that all the medium-sensitivity threads' needs are satisfied. Recall that we adopt the Android priority system for thread prioritization. Let the number of computing cycles allocated to the medium-sensitivity threads be C_m in this sampling period. Then, the number of cycles allocated to low-sensitivity threads on the same core can be calculated by $C_\ell = C_m \times \frac{N_\ell}{N_\ell + 1.25^k \times N_m}$, where N_ℓ and N_m denote the respective numbers of low- and medium-sensitivity threads on the core; and k is the difference between their nice values. Therefore, the core should operate on a frequency of at least $(C_\ell + C_m)/0.2$ MHz if we want to satisfy the medium-sensitivity threads' needs.

Commercial mobile devices allow operations on a number of frequency levels. We could simply select the lowest available level that is just sufficient to satisfy the cycles calculated above. However, such a selection could not react promptly to sudden workload increases. Thus, a higher level should be selected if the workload is likely to be heavier than expected. The governor selects one level higher than the originally selected level if the the original level could result in core utilization above a certain threshold[10]. Furthermore, if the workload increases twice in a row, the governor selects the median between the current level and the highest level to deal with a potential bursty workload. The half-scaling policy could scale up to the required level quickly without wasting much energy because the CPU power model is an exponential function of the frequency level. Finally, the highest among the levels selected for the active cores is used if the CPU only supports synchronous scaling. Note that the big and LITTLE clusters are supposed to support frequency levels in different ranges; thus, the frequency level is selected from the union of all the levels of the two clusters. Moreover, the frequency level will remain the same during the sampling period unless some event, like the user touching the screen, changes the sensitivity of any threads to high.

Cluster Switching: Once the operating frequency has been selected, the governor determines whether to switch the CPU from one cluster to the other. If the LITTLE cluster is the currently used cluster and the selected frequency exceeds its highest level, the big cluster is activated immediately with the same number of cores being turned on; then, the threads running on each LITTLE core are moved to a corresponding big core. Cluster switching is performed immediately so as to ensure the quality of user experience. In contrast, if the big cluster is used currently and the selected frequency is below its lowest level, the cluster will remain unchanged (and thus its lowest level is used instead), unless this has also happened in the preceding two sampling periods. Recall that the half-scaling policy will be trigged if the workload increases twice in a row, as mentioned in the DVFS design. In other words, if the selected frequency continues relatively low, the workload is likely to be light so that the LITTLE cluster will be able to cope. Accordingly, the LITTLE cluster is activated to save power. Moreover, the overhead incurred by cluster switching is relatively large, compared with that required for DPM. Such a design can alleviate the ping-pong effect between the clusters as well.

3.3. Implementation Issues

In this section, we consider some technical issues that arise when implementing our scheduler and governor in the Android operating system.

[10]Our implementation sets the threshold at 75%, which is the average of the two corresponding thresholds, 90% and 60%, set respectively for the big and LITTLE clusters in the Ondemand implementation of Samsung Galaxy S4.

Integration Details: For portability across different Android platforms, we implement most of our design in the user space, and minimize the modifications of kernel codes by adding several hooks in the kernel space. The communications between the kernel and the user space are handled by *proc connector* [Love 2010]. Since the native proc connector can only pass thread events, we extend it to pass hardware driver events as well. We add two hooks to the *kernel core* to trigger an event when a thread is forked or terminated; and one hook to the *touchscreen driver* and another to the *display driver* to trigger corresponding events because these events will be recorded by the corresponding hardware drivers in the kernel layer. Furthermore, we create a kernel module comprised of two timers: one generates an event every 200 ms to indicate a sampling period, while the other generates an event when the count reaches 500 ms after each user interaction. The decisions made by the scheduler and governor in the user space, like a thread's movement and the operating frequency, are communicated to the kernel via system calls and *sysfs* [Love 2010]. Our implementation comprises 25 files and 3879 lines of C code, among which 400 lines are scattered in 4 files belonging to the kernel space.

Our scheduler and governor react to each of the six events as follows. 1) On receipt of an event where a thread is forked, the scheduler performs thread allocation and thread prioritization to determine the thread's core and nice value. 2) If a termination event occurs, all the information about the thread maintained by the scheduler is deleted. 3) For a touch event, the scheduler performs thread prioritization to assign the foreground threads the nice value that corresponds to high sensitivity, and then resets the 500ms timer to record the elapsed time. In addition, the governor performs DVFS to adjust the operating frequency to the highest level. 4) If the event is triggered by the display toggle, the scheduler performs thread prioritization to reflect the thread transitions caused by turning the display on/off. 5) An event generated periodically by the 200ms timer invokes DPM, thread migration, DVFS, and cluster switching in sequence. Moreover, whenever such an event is generated five times[11], the scheduler queries the application manager in the framework layer about the current foreground application via *dumpsys*, and performs thread prioritization if necessary. 6) When the 500ms timer expires, the high-sensitivity threads revert to medium sensitivity.

Workload Estimation: The total workload of a core in a sampling period is normally defined as the sum of the computing cycles used by the threads running on the core. Note that we cannot determine how many cycles a thread will eventually use in a sampling period during its execution. A general practice, as also adopted by Samsung Galaxy S4, uses the historical workload in the previous sampling period as the basis for workload estimation. This estimation strategy is sensible for mobile applications because of its adaptability to changeable workloads. Similarly, to estimate the workload of each running thread, we monitor how many computing cycles the thread used in the previous sampling period and then estimate its computing cycles in the current period. In Android, the amount of time that every thread has been executed is maintained by the kernel (specifically, in /proc) since it is forked. Therefore, the number of computing cycles used by a thread in a sampling period can be calculated by simply multiplying (i) the ratio of the thread's execution time in the period to the length of the sampling period by (ii) the CPU frequency used for the period.

Extra Overheads: Our design distinguishes between threads according to how they may affect user experience. Hence, in every sampling period, we have to update the

[11]In Android, all the running applications are maintained by the application manager in the framework layer, while all threads belonging to an application are maintained by the kernel (or, more precisely, in /proc). The communication between the application manager and the kernel requires some time because it involves two languages, namely Java and C.

sensitivity and estimate the computing cycles of each thread, as well as search for the threads associated with some specific sensitivity. These fine-grained actions introduce extra overheads in our design, compared with traditional coarse-grained approaches that only consider the number of running threads and the utilization of each core. To reduce the search and computation overheads, in addition to maintaining the threads in some elaborate data structures, we set the sampling period at 200 ms to enable a trade-off between the extra overheads and the reaction time to workload changes. Our sampling period is longer than that adopted in traditional approaches; for example, Samsung Galaxy S4 sets the sampling period at 10 ms for thread migration, and 100 ms for DPM along with DVFS and cluster switching.

4. PERFORMANCE EVALUATION

4.1. Experiment Setup

To gain further insights into user-centric scheduling and governing, we conducted extensive experiments on a commercial Android smartphone with some real-world mobile apps. We used a Samsung Galaxy S4 smartphone, which is equipped with two different quad-core processors. The LITTLE cluster allows operations on 8 frequency levels in the range 250 to 600 MHz, while the big cluster supports 9 frequency levels in the range 800 to 1600 MHz. Either cluster can be active at a time, and the four cores of the active cluster have to operate on the same frequency synchronously, but they can be turned on/off individually. The specifications of the related hardware and software are detailed in Table I. If necessary, the smartphone can access the Internet via an ASUS RT-N10 802.11n access point dedicated for our experiments. We used the power monitor produced by Monsoon Solutions[12] to measure the smartphone's transient power and energy consumption. In addition, we instrumented the source codes of the investigated apps to measure their response and completion times (if they did not provide such information). The experiment environment is shown in Figure 4.

Table I. Specifications of Samsung Galaxy S4.

Hardware	
CPU	Quad-core ARM Cortex-A15 (800∼1600 MHz, 9 levels) Quad-core ARM Cortex-A7 (250∼600 MHz, 8 levels)
Memory	2GB LPDDR3 RAM
Screen	Super AMOLED 1920×1080 pixels
Network	IEEE 802.11 a/b/g/n/ac WiFi
Storage	16GB SD 2.0 compatible
Battery	2600 mAh
Software	
OS	Android 4.2.2 Linux Kernel 3.4.5

We studied possible combinations of applications with different characteristics, namely, interactive, non-interactive, CPU-intensive, and I/O-intensive. OI File Manager, RockPlayer, OpenExplorer, and FtpCafe, all of which can be found on Google Play, were chosen for the performance evaluation[13]. OI File Manager provides a user

[12]Monsoon Solutions, Inc., http://www.msoon.com.

[13]It would be ideal to use popular apps. However, to ensure a fair comparison, the input workloads for different designs must be reproducible and the performance of the investigated apps should be quantifiable. Thus, the source codes of the apps should be available for instrumentation. The four apps were chosen because they can generate respective workload patterns of the four application types. The reported results may be similar to the results of experiments conducted based on some popular apps without open source codes, because different types of applications usually have different workload patterns but the same type may have similar patterns.

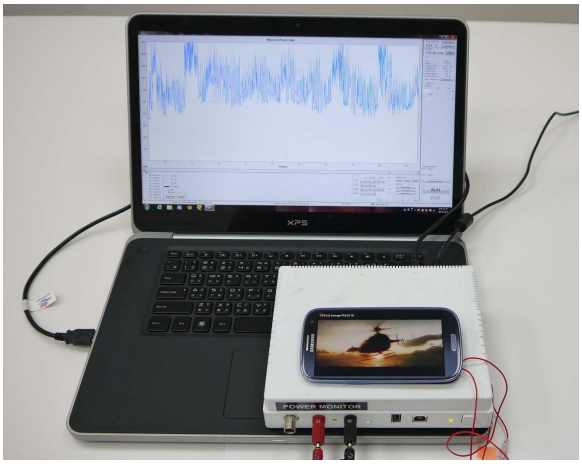


Fig. 4. The experiment environment.

interface to manage files and directories on a smartphone. It served as an interactive app, and was used to open and close a directory (which took 1s and 0.5s, respectively) 10 times to generate high-sensitivity threads. RockPlayer is a software-decoding video player for playing multimedia files. It was deemed a non-interactive app, and used to play a 60-second movie trailer with a resolution of 640×360 and a frame rate of 30 fps so as to create long-running threads with medium sensitivity. OpenExplorer provides a zip facility for file compressing and uncompressing. It served as a CPU-intensive app, and was used to compress a 67.4MB file to create low-sensitivity threads. FtpCafe is an FTP client that enables file transfer from one host to another. It served as an I/O-intensive app, and was used to download a 65.9MB file from the Internet to create low-sensitivity threads. The investigated apps are categorized in Table II. OI File Manager and RockPlayer were used as foreground applications, while OpenExplorer and FtpCafe were used as background applications. Note that, to reduce the potential influence of human intervention, we wrote scripts to launch the apps and trigger the corresponding actions in all the experiments.

Table II. Mobile Apps Used in the Experiments.

Foreground Apps		Background Apps	
Interactive	OI File Manager	CPU-intensive	OpenExplorer
Non-interactive	RockPlayer	I/O-intensive	FtpCafe

We compared our user-centric scheduler and governor (denoted as UCSG) with a conventional design (denoted as CFS+OD). Recall that we discussed the implementation and tunable parameters of UCSG in Section 3. CFS+OD employs the CFS [Love 2010] as the scheduler to achieve fair scheduling and implements the governor extended based on *Ondemand* [Pallipadi and Starikovskiy 2006] for power management, as described in Section 2.1. To show the efficacy of UCSG and CFS+OD in energy reduction, the adopted metric was the total energy consumption of the CPU required for a scenario (i.e., launching and executing some combination of apps). Furthermore, for the performance comparison, different apps should have appropriate metrics. The metrics adopted for the file manager and the video player were the response time and

the frame rate respectively; while the metric for the ZIP and the FTP was the completion time. To assess the extra overheads incurred by UCSG, we also considered the scenarios where the smartphone is idle with the screen switched on or off.

4.2. Computational Overheads

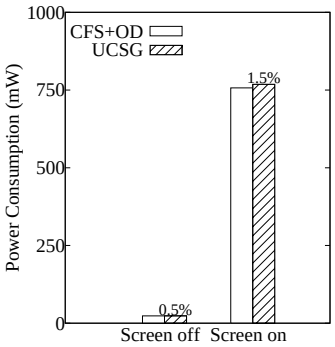


Fig. 5. Idle power required by CFS+OD and UCSG.

Figure 5 shows the power consumption when the mobile device is idle. When the screen is turned off, the whole CPU is put into a low-power dormant mode; thus, the device consumes hardly any power. When the screen is turned on, one of the four cores of the LITTLE cluster is active and operates on the lowest frequency of 250 MHz, and the UCSG's power consumption is slightly higher than that of CFS+OD. This is because, compared with traditional core-level monitoring, the thread-level monitoring in our design introduces extra computational overheads. Specifically, in every sampling period, UCSG has to update the sensitivity and estimate the computing cycles of each thread. As discussed in Section 3.3, these fine-grained actions introduce extra overheads, compared with traditional coarse-grained approaches that only consider the number of running threads and the utilization of each core. However, this scenario is relatively short in terms of mobile users' daily usage patterns, as the screen usually switches off automatically (or is turned off by the user) to save energy when the device is idle.

4.3. Energy Consumption

Figure 6 shows the energy required by CFS+OD and UCSG to execute various combinations of applications. In general, UCSG requires significantly less energy than CFS+OD. This result is as expected because UCSG only allocates limited resources to low-sensitivity threads. However, in the scenarios when no applications run in the background, UCSG can still reduce the energy consumption of CFS+OD by 31.4% for the file manager and 15.2% for the video player, because of its governor design (especially with the half-scaling policy). The efficacy of UCSG becomes clearer, in general, when more applications are running in the background. The results show that UCSG reduces the energy consumption under CFS+OD by 26.2%, 22.6%, and 24.9% respectively when the file manager runs in the foreground while ZIP, FTP, and both ZIP and FTP run in the background. Comparably, the same combinations of background applications yield 22%, 11.5%, and 23.4% energy reduction when the video player runs in the foreground. Interestingly, the reduction when FTP, an I/O-intensive application, runs in the background is as significant as the reduction when ZIP, a CPU-intensive application, runs in the background. The reason is that, although FTP's CPU workloads

are light, it generates workloads intermittently. Moreover, CFS+OD always scales the operating frequency to the highest level to deal with bursty workloads, while UCSG decides whether to scale up the frequency based on the thread sensitivity.

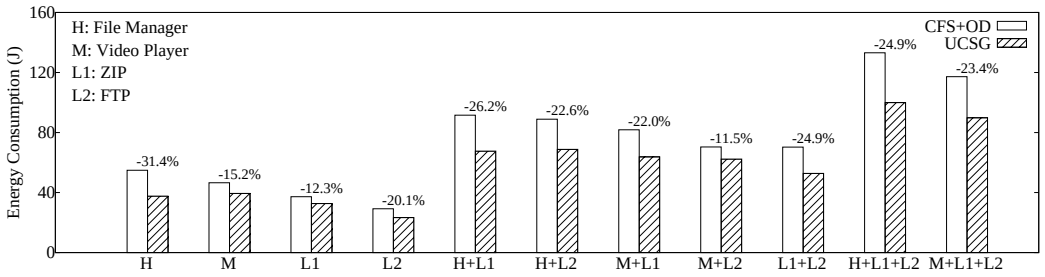


Fig. 6. Energy consumption required by CFS+OD and UCSG.

For the scenarios where ZIP, FTP, and both ZIP and FTP run in the background only, UCSG still achieves significant energy savings and reduces the energy consumed under CFS+OD by 12.3% for ZIP, by 20.1% for FTP, and by 24.9% when both run simultaneously. In these three scenarios, the foreground application is the home user interface, whose CPU workload is light if there are no user interactions. Consequently, UCSG normally uses the LITTLE cluster operating on the lowest frequency of 250 MHz during the whole execution, unless the workload increases twice in a row, which would trigger the half-scaling policy. This explains the considerable reduction in energy consumption. It is noteworthy that the reduction is greater for FTP than for ZIP because of FTP's intermittent CPU workloads. Furthermore, the reduction is more obvious when the file manager runs in the foreground than when the video player runs. This is because the former generates CPU workloads that are shorter and vary significantly, while the latter's workloads are more stable and consistent. When bursty workloads occur, the half-scaling policy of UCSG can scale up to the required frequency level quickly, without wasting much energy. The results show that UCSG reduces the energy consumed under CFS+OD between 11.5% and 31.4% in various application scenarios.

4.4. Application Performance

Figure 7 shows the four applications' respective performance achieved by CFS+OD and UCSG. As can be seen in Figure 7(a), UCSG greatly outperforms CFS+OD in terms of the file manager's response time. The reason that UCSG reduces the response time required under CFS+OD by 25.9% to 35.2% is that UCSG scales the frequency to the highest level immediately when a thread changes to the high sensitivity state. CFS+OD, on the other hand, scales to the highest level in the next sampling period if it senses potential workload increases in the current period. This also explains why the response time achieved by UCSG is generally stable, while that required under CFS+OD varies significantly with different applications running in the background. As shown in Figure 7(b), the average frame rates achieved by both CFS+OD and UCSG are close to the frame rates required for video playing. CFS+OD slightly outperforms UCSG because (1) the available CPU resources are sufficient to sustain the investigated scenarios; and (2) CFS+OD always scales up to the highest frequency to react to any abrupt workload changes, while UCSG employs a half-scaling policy. Accordingly, CFS+OD attempts to ensure the application performance at the cost of increased

power consumption. By contrast, UCSG tends to save more power, provided that the application performance is guaranteed. We explain their difference in more detail later with the CPU's footprints shown in Figure 8. However, both CFS+OD and UCSG can decode over 29 frames per second in all the scenarios; that is, the video can be played smoothly.

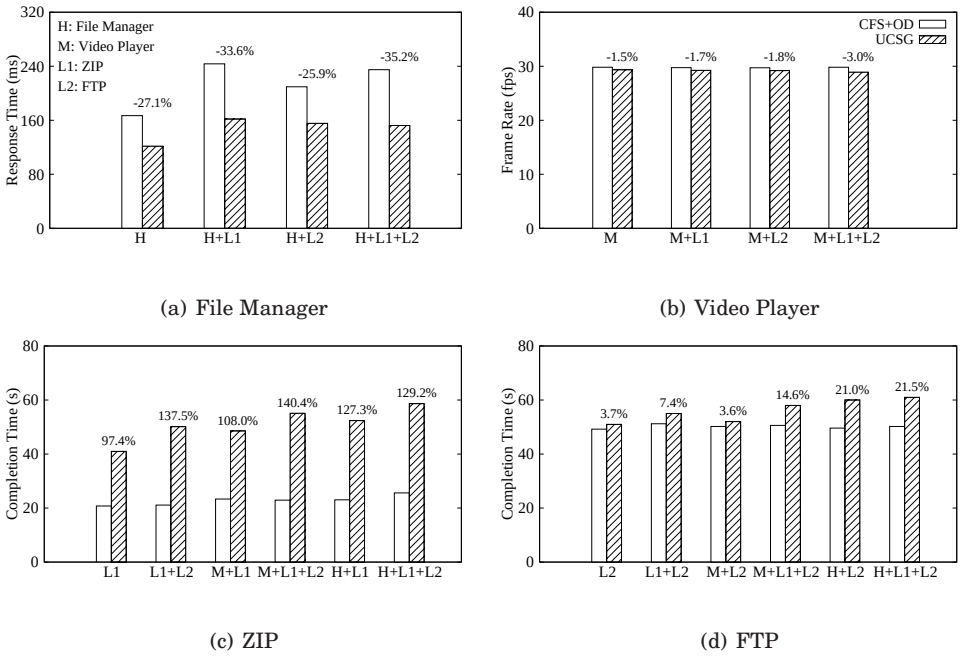
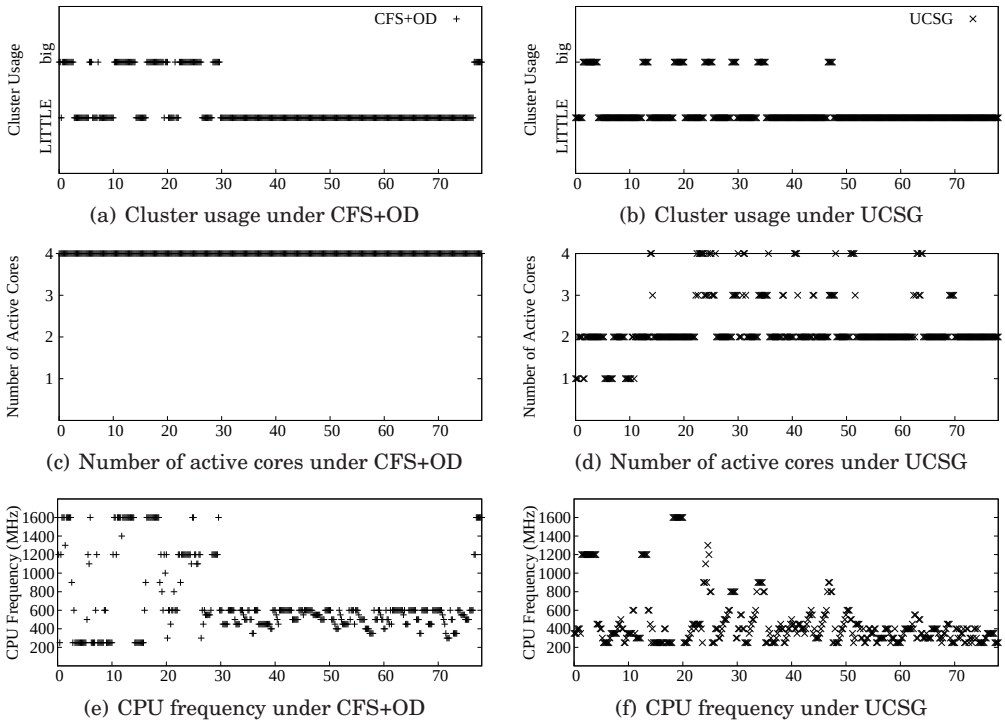


Fig. 7. Application Performance achieved by CFS+OD and UCSG.

Figure 7(c) shows the completion times of ZIP under CFS+OD and UCSG in different scenarios. As expected, CFS+OD outperforms UCSG because the latter only provides background applications with limited computing resources in order to save energy. In particular, when the foreground application has heavy workloads and/or some other application running in the background shares the limited computing resources, the completion time is prolonged further. The results show that UCSG increases the completion time required under CFS+OD by between 97.4% and 140.4% for different scenarios. The significant increase is due to the extreme parameter settings used in our UCSG implementation, and can be reduced by changing the settings of nice values in thread prioritization. Figure 7(d) shows the completion times of FTP under CFS+OD and UCSG. The results show that UCSG increases the completion time required under CFS+OD by between 3.7% and 21.5% when FTP is only provided with limited computing resources. CFS+OD outperforms UCSG more significantly when the file manager runs in the foreground than when the video player runs. This is because FTP is allocated an even smaller proportion of the CPU time per scheduling period when high-sensitivity threads exist. Furthermore, the increase is relatively small, compared with the increase in ZIP's completion time, because FTP is not a CPU-intensive application and its completion time is highly dependent on the network bandwidth.

4.5. CPU Footprints

Figure 8 shows the CPU footprints, i.e., the changes in the cluster usage, the number of active cores, and the operating frequency, under CFS+OD and UCSG when the video player runs in the foreground while ZIP is executed in the background. Note that the whole period of this experiment includes the time required to launch the apps and change directories to select the target files; moreover, the corresponding actions are deemed high-sensitivity threads in UCSG. As shown in Figure 8(a), CFS+OD switches the cluster frequently between the big and the LITTLE until the 30th second. This is because ZIP is a CPU-intensive application, so the big cluster is activated whenever the LITTLE one is unable to cope. We observed that ZIP finishes by the 24th second but CFS+OD does not react to the workload change immediately. On the other hand, UCSG uses the big cluster occasionally before ZIP finishes at the 49th second, as shown in Figure 8(b), because it limits the computing resources allocated to ZIP. As far as the whole period is considered, CFS+OD and UCSG use the big cluster for 23.8% and 14.3% of the time, respectively.



(a) Cluster usage under CFS+OD (b) Cluster usage under UCSG

(c) Number of active cores under CFS+OD (d) Number of active cores under UCSG

(e) CPU frequency under CFS+OD (f) CPU frequency under UCSG

Fig. 8. CPU footprints under CFS+OD and UCSG for M+L1.

With regard to the number of active cores, Figure 8(c) shows that CFS+OD always uses four cores for the whole period. This is because CFS+OD turns all the cores on once the operating frequency exceeds a threshold (which is set at the current cluster's lowest level, i.e., 250 or 800 MHz, in Samsung Galaxy S4), and will not turn a core off unless the frequency reaches the same threshold and remains at that frequency for a few sampling periods (which is set at five). In contrast, the number of active cores under UCSG is usually two, but varies over time, as shown in Figure 8(d). The reason is that UCSG turns cores on or off based on the total workload in each sampling period

and only provides ZIP with limited computing resources. The number of active cores under UCSG for the whole period is about 2.2 on average.

As shown in Figure 8(e), the operating frequency under CFS+OD varies substantially between 250 and 1600 MHz (i.e., the big's highest level) until the 30th second. The frequency is then scaled down and generally remains between 400 and 600 MHz (i.e., the LITTLE's highest level) for the last 48 seconds until the video finishes. This is because CFS+OD scales the frequency directly to the highest level when a core's utilization exceeds a predefined threshold (which is set respectively at 60% and 90% for the LITTLE and big clusters in Samsung Galaxy S4), and it scales down the frequency step by step when the utilization of all active cores falls below another threshold (which is set at 40% and 70% for the LITTLE and big clusters, respectively). In contrast, Figure 8(f) shows that UCSG generally uses frequency levels lower than those used by CFS+OD, except when the actions deemed high-sensitivity threads are triggered. The average frequency used by UCSG is 470 MHz, while that used by CFS+OD is 704 MHz. In summary, UCSG uses the LITTLE cluster for most of the execution and fewer cores than CFS+OD at lower frequency levels; hence, it is more energy-efficient.

5. RELATED WORK

In this section, we provide a comprehensive review of related work from an application-driven perspective.

Real-Time Energy-Efficient Scheduling: Yao et al. [Yao et al. 1995] designed a scheduling model that executes tasks on a single variable-speed CPU. They also investigated how the energy dissipation of the CPU can be reduced by adjusting its operating speed dynamically. Their pioneer work has spawned extensive studies on DVFS, and many theoretical results have been published for *real-time applications* with various system and power models under different priority-driven scheduling algorithms to ensure that such applications can be executed without violating their deadline requirements, while minimizing the energy consumption [Aydin et al. 2006; Chen et al. 2005; Ishihara and Yasuura 1998; Jejurikar and Gupta 2004; Quan and Hu 2007]. The advent of multi-core CPUs has made DVFS more complicated [Yang et al. 2005; Zhang et al. 2002], and also brought the research issue on DPM [Benini et al. 2000], which tries to switch cores to a low-power state when they are expected to be idle for sufficiently long periods. For the combination of DVFS and DPM, some interplay between the two techniques should be considered in order to minimize the energy consumption [Devadas and Aydin 2012; Gerards and Kuper 2013]. The above research provides rigid theoretical analysis of real-time applications whose execution behavior, such as their worst-case execution times and deadlines, is highly predicable or known a priori.

Application-Specific DVFS and DPM: The popularity of mobile devices equipped with DVFS-enabled CPUs has motivated another research direction for *mobile applications*. Multimedia playing is a major mobile application, and estimating the decoding times of video frames accurately is the core issue when performing DVFS [Hamers and Eeckhout 2012; Pouwelse et al. 2001]. For 3D graphics games, signature-based estimation was proposed to obtain frame signatures from the graphics engine for workload prediction [Mochocki et al. 2006]; and in [Gu and Chakraborty 2008], the frame structure of graphics games was exploited to improve the prediction accuracy. For interactive applications, user-perceived latency acquired from the users' feedback was utilized to deal with the dynamic nature of user behavior [Bi et al. 2010; Yan et al. 2005]. In addition, DVFS techniques have been developed especially for streaming applications on multi-core CPUs [Singh et al. 2013], and DPM techniques have been developed for sensing applications [Herrmann et al. 2012]. The above techniques were designed for specific applications and achieve considerable energy savings by leveraging the applications' special characteristics.

Application-Agnostic DVFS and DPM: Recently, mobile applications have become more popular and diverse. As users often install various applications that have specific purposes, the design issues of DVFS and DPM become more challenging. AutoDVS [Gurun and Krintz 2005] was developed as a general-purpose DVFS scheme that can distinguish between interactive and batch sessions, and then apply different scaling policies to each session type. Based on the observation that different applications may have different usage patterns on hardware resources, a resource-driven DVFS scheme was proposed in [Chang et al. 2013] to explore the interplay between the CPU and other resources to facilitate frequency scaling. The *Ondemand* governor [Pallipadi and Starikovskiy 2006], which was introduced with Linux 2.6, is used by Android as the default DVFS scheme. The above schemes were all designed for single-core CPUs. For DPM and cluster switching, device vendors may have different implementations. However, there has been comparatively little research on application-agnostic DVFS, DPM, and cluster switching (i.e., governing) for mobile systems, particularly in conjunction with thread scheduling.

6. CONCLUDING REMARKS

We advocate that mobile operating systems should move toward user-centric scheduling and governing, in which computing resources are allocated to applications in proportion to the degrees of attention they receive from the user. To demonstrate the benefits, we devised a scheduler and governor that allocate computing resources to mobile applications according to their sensitivity, and implemented our design in the Android operating system. In addition, we conducted a series of experiments on a Samsung Galaxy S4 smartphone with some mobile apps found on Google Play. The experiment results show that our user-centric design is more appropriate for mobile applications, compared with conventional fair scheduling and governing. The proposed design is particularly suitable when some applications running in the background generate bursty CPU workloads intermittently. This is because our design can identify abrupt workload changes caused by applications with different levels of sensitivity, and determine whether to increase the computing resources for certain applications.

In the future, we will consider more sensitivity levels and investigate the impacts of the number of levels, as well as other tunable parameters, on the efficacy of user-centric scheduling and governing. We will also extend and accommodate our design to heterogeneous multi-core architectures with more advanced and flexible features.

REFERENCES

ABENI, L. AND BUTTAZZO, G. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of IEEE RTSS*. 4–13.

ABENI, L. AND BUTTAZZO, G. 1999. Adaptive Bandwidth Reservation for Multimedia Computing. In *Proc. of IEEE RTCSA*. 70–77.

AYDIN, H., DEVADAS, V., AND ZHU, D. 2006. System-Level Energy Management for Periodic Real-Time Tasks. In *Proc. of IEEE RTSS*. 313–322.

BENINI, L., BOGLIOLO, A., AND DE MICHELI, G. 2000. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Trans. on VLSI Systems 8,* 3, 299–316.

BI, M., CRK, I., AND GNIADY, C. 2010. IADVS: On-demand Performance for Interactive Applications. In *Proc. of IEEE HPCA*. 1–10.

CHANDRA, A., ADLER, M., AND SHENOY, P. 2001. Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *Proc. of IEEE RTAS*. 3–14.

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. on Embedded Computer Systems 13,* 3, 53:1–53:22.

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 *Proc. of ACM EMSOFT*. 247–250.

CUCINOTTA, T., PALOPOLI, L., MARZARIO, L., LIPARI, G., AND ABENI, L. 2004. Adaptive Reservations in a Linux Environment. In *Proc. of IEEE RTAS*. 238–245.

DEVADAS, V. AND AYDIN, H. 2012. On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-Based Real-Time Embedded Applications. *IEEE Trans. on Computers 61,* 1, 31–44.

DONOHOO, B. K., OHLSEN, C., AND PASRICHA, S. 2011. AURA: An Application and User Interaction Aware Middleware Framework for Energy Optimization in Mobile Devices. In *Proc. of IEEE ICCD*. 168–174.

FALAKI, H., MAHAJAN, R., AND KANDULA, S. 2010. Diversity in Smartphone Usage. In *Proc. of ACM MobiSys*. 179–193.

GERARDS, M. E. T. AND KUPER, J. 2013. Optimal DPM and DVFS for Frame-Based Real-Time systems. *ACM Trans. on Architecture and Code Optimization 9,* 4, 1–23.

GORANSSON, A. 2014. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications* 1st Ed. O'Reilly, 29–37.

GU, Y. AND CHAKRABORTY, S. 2008. Control Theory-Based DVS for Interactive 3D Games. In *Proc. of IEEE/ACM DAC*. 740–745.

GURUN, S. AND KRINTZ, C. 2005. AutoDVS: An Automatic, General-Purpose, Dynamic Clock Scheduling System for Hand-Held Devices. In *Proc. of IEEE/ACM EMSOFT*. 218–226.

HAMERS, J. AND EECKHOUT, L. 2012. Exploiting Media Stream Similarity for Energy-Efficient Decoding and Resource Prediction. *ACM Trans. on Embedded Computing Systems 11,* 1, 2:1–2:25.

HERRMANN, R., ZAPPI, P., AND ROSING, T. S. 2012. Context Aware Power Management of Mobile Systems for Sensing Applications. In *Proc. of International Workshop on Mobile Sensing*.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proc. of IEEE/ACM ISLPED*. 197–202.

JEJURIKAR, R. AND GUPTA, R. 2004. Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems. In *Proc. of IEEE/ACM ISLPED*. 78–81.

LI, T., BAUMBERGER, D., AND HAHN, S. 2009. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin. In *Proc. of ACM SIGPLAN PPoPP*. 65–74.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of the ACM 20,* 1, 46–61.

LOVE, R. 2010. *Linux Kernel Development* 3rd Ed. Addison-Wesley Professional, 48–50, 355–357, 361.

MAIA, C., NOGUEIRA, L., AND PINHO, L. M. 2010. Evaluating Android OS for Embedded Real-Time Systems. In *Proc. of OSPERT*. 63–70.

MERCATI, P., BARTOLINI, A., PATERNA, F., ROSING, T. S., AND BENINI, L. 2013. Workload and User Experience-Aware Dynamic Reliability Management in Multicore Processors. In *Proc. of IEEE/ACM DAC*. 1–6.

MERCATI, P., BARTOLINI, A., PATERNA, F., ROSING, T. S., AND BENINI, L. 2014. A Linux-Governor Based Dynamic Reliability Manager for Android Mobile Devices. In *Proc. of IEEE/ACM DATE*. 1–4.

MOCHOCKI, B. C., LAHIRI, K., CADAMBI, S., AND HU, X. S. 2006. Signature-Based Workload Estimation for Mobile 3D Graphics. In *Proc. of IEEE/ACM DAC*. 592–597.

MUTAPCIC, A., BOYD, S., MURALI, S., ATIENZA, D., DE MICHELI, G., AND GUPTA, R. 2009. Processor Speed Control With Thermal Constraints. *IEEE Trans. on Circuits and Systems I: Regular Papers 56,* 9, 1994–2008.

PALLIPADI, V. AND STARIKOVSKIY, A. 2006. The Ondemand Governor: Past, Present, and Future. In *Proc. of Linux Symposium*. Vol. 2. 223–238.

POUWELSE, J., LANGENDOEN, K., AND SIPS, H. 2001. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *Proc. of ACM MobiCom*. 251–259.

QUAN, G. AND HU, X. 2007. Energy Efficient DVS Schedule for Fixed-Priority Real-Time Systems. *ACM Trans. on Embedded Computing Systems 6,* 4, 29:1–29:31.

SEGOVIA, V. R., ÅRZÉN, K.-E., SCHORR, S., GUERRA, R., FOHLER, G., EKER, J., AND GUSTAFSSON, H. 2010. Adaptive Resource Management Framework for Mobile Terminals - the ACTORS Approach. In *Proc. of WARM*.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2010. *Operating System Concepts* 8th Ed. John Wiley & Sons, 196–199.

SINGH, A. K., DAS, A., AND KUMAR, A. 2013. Energy Optimization by Exploiting Execution Slacks in Streaming Applications on Multiprocessor Systems. In *Proc. of IEEE/ACM DAC*. 1–7.

TOLIA, N., ANDERSEN, D. G., AND SATYANARAYANAN, M. 2006. Quantifying Interactive User Experience on Thin Clients. *IEEE Trans. on Computer 39,* 3, 46–52.

TSENG, P.-H., HSIU, P.-C., PAN, C.-C., AND KUO, T.-W. 2014. User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices. In *Proc. of IEEE/ACM DAC*. 1–6.

WEI, Y.-H., YANG, C.-Y., KUO, T.-W., HUNG, S.-H., AND CHU, Y.-H. 2010. Energy-Efficient Eeal-Time Scheduling of Multimedia Tasks on Multicore Processors. In *Proc. of ACM SAC*. 258–262.

YAN, L., ZHONG, L., AND JHA, N. K. 2005. User-Perceived Latency Driven Voltage Scaling for Interactive Applications. In *Proc. of IEEE/ACM DAC*. 624–627.

YANG, C.-Y., CHEN, J.-J. C., AND KUO, T.-W. 2005. An Approximation Algorithm for Energy-Efficient Scheduling on A Chip Multiprocessor. In *Proc. of IEEE/ACM DATE*. 468–473.

YAO, F., DEMERS, A., AND SHENKER, S. 1995. A Scheduling Model for Reduced CPU Energy. In *Proc. of IEEE FOCS*. 374–382.

ZHANG, Y., HU, X. S., AND CHEN, D. Z. 2002. Task Scheduling and Voltage Selection for Energy Minimization. In *Proc. of IEEE/ACM DAC*. 183–188.