Phase-Aware Web Browser Power Management on HMP Platforms

N. Peters[1], S. Park[1], D. Clifford[2], S. Kyostila[2], R. McIlroy[2], B. Meurer[2], H. Payer[2], S. Chakraborty[1]
Technical University of Munich[1], Google Inc[2]
{nadja.peters,sangyoung.park,samarjit}@tum.de,{danno,skyostil,rmcilroy,bmeurer,hpayer}@google.com

ABSTRACT

Over the last years, web browsing has been steadily shifting from desktop computers to mobile devices like smartphones and tablets. However, mobile browsers available today have mainly focused on performance rather than power consumption, although the battery life of a mobile device is one of the most important usability metrics. This is because many of these browsers have originated in the desktop domain and have been ported to the mobile domain. Such browsers have multiple power hungry components such as the rendering engine, and the JavaScript engine, and generate high workload without considering the capabilities and the power consumption characteristics of the underlying hardware platform. Also, the lack of coordination between a browser application and the power manager in the operating system (such as Android) results in poor power savings. In this paper, we propose a power manager that takes into account the internal state of a browser – that we refer to as a *phase* – and show with Google's Chrome running on Android that up to 57.4% more energy can be saved over Android's default power managers. We implemented and evaluated our technique on a heterogeneous multi-processing (HMP) ARM big.LITTLE platform such as the ones found in most modern smartphones.

KEYWORDS

Android Power Management; DVFS; CPU Frequency Governor; Heterogeneous Multi-Processing; big.LITTLE; Mobile Web Browser

1 INTRODUCTION

Mobile devices such as smartphones and tablets have become an integral part of our daily lives. The time we spend on such devices has recently surpassed the time spent on desktop computers [29]. Using mobile devices for daily activities like instant messaging, social networking, and web browsing always involves being connected to the Internet. Although a large variety of mobile applications exists, one of the most traditional, and one of the most preferred, continues to be the web browser [18]. Therefore, ensuring the quality of the user experience during mobile web browsing is an important problem.

The user experience of mobile web browsers is multi-faceted. For example, users are sensitive to the responsiveness of the screen to touch events, e.g., zooming or scrolling web pages [21, 43]. Hence, modern web browsers usually target a frame rate of 60 frames per second (FPS) to guarantee a good user experience. Further, when loading a web page, users want to view the page within a reasonable amount of time. Meanwhile, web pages and consequently also web browsers have grown increasingly complex along with increasing network and hardware computation speed. Given these trends, web browser performance has always been boosted in order to ensure good quality of the user experience. Browsers have developed into complex applications that consist of multiple components such as a browser engine, a rendering engine and a JavaScript engine [15]. Such complexity, and also the focus on performance are because many mobile browsers have evolved from the desktop world.

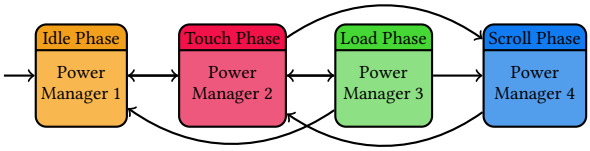

Figure 1: Proposed phase-aware power management.

However, high performance comes at the cost of higher power consumption (e.g., by the underlying CPU). While this is not a critical drawback for desktop computers, it is a much more serious issue in battery-constrained mobile devices. For such devices, the battery lifetime is one of the most important usability factors to which users now pay a significant amount of attention [36]. While power management of mobile devices has been extensively studied for several years now, the focus has been on video decoding applications [24, 38], and also on games [13, 25]. Surprisingly, power management techniques specifically targeting the web browser have been less studied [35, 45, 48]. But both, its importance, and its potential for power savings, is increasingly being recognized.

Android power management on HMP platforms: While a mobile device comprises a multitude of power hungry components such as the display or the wireless link, our work focuses on CPU power consumption. Recent work has shown, that the CPU power consumption contributes on average 38% towards the daily energy drainage of a smartphone [6]. In general, the overall phone power consumption improvement heavily depends on the usage scenario and the power management strategies of the other components as well, not only the CPU. E.g. if the screen is very bright, it will consume more power whereas the whole phone power savings would be less compared to the case where the display is dimmed.

CPU Power management in Android is implemented through CPU frequency governors, thread schedulers and wakelock mechanisms in the operating system (OS). They respectively determine at which voltage and frequency the CPUs operate, which threads run on which CPU cores and which CPU cores are active. The most popular governors in Android are the *ondemand* [33] and the *interactive* [5] governors. They adjust the operating CPU frequency (*dynamic voltage and frequency scaling*, *DVFS*) according to the CPU's utilization, with a focus on the responsiveness to user inputs

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06.
https://doi.org/10.1145/3205289.3205293

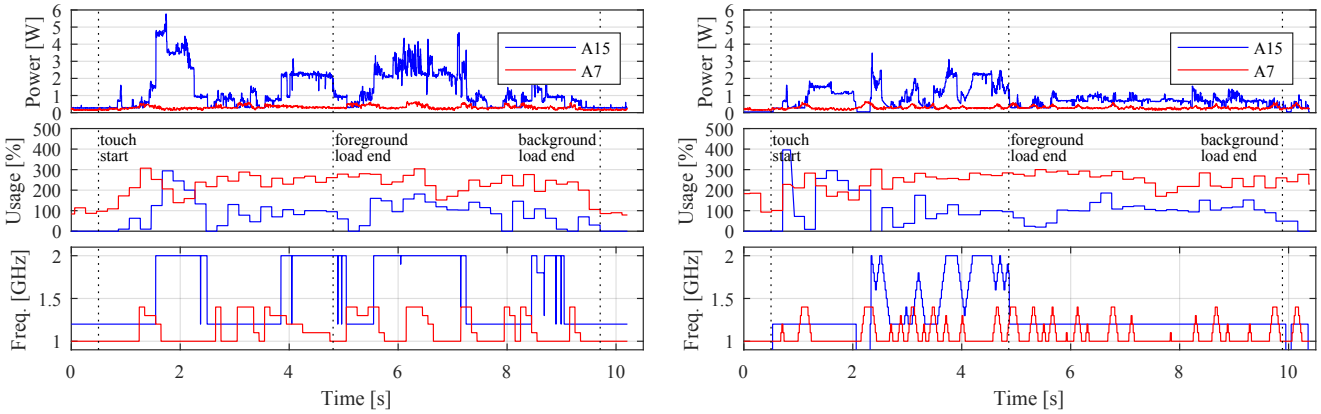

Figure 2: Power, CPU usage and CPU frequency for loading the Reddit page with ondemand (left) and browser governor (right).

by ramping up the frequency quickly and conservatively reducing it. Also, they do not power down (*power gate*) CPU cores during runtime as the penalty for turning the cores back on affects the system responsiveness. Even though such governors perform well with power management over a wide variety of applications, they have fundamental limitations. A major limitation in achieving optimal power management comes from the fact that the user application and the OS power management entities are highly modularized, and lack communication channels among them. In particular, this results in energy wastage on heterogeneous multi-processing (HMP) system-on-chips (SoCs) with big.LITTLE architectures, which are widely used in modern smartphones, such as the Nexus 5X (Snapdragon 808) [17, 37], the Samsung Galaxy S8 (Exynos 8895) [39] and even the iPad pro [42].

The Android default governors monitor the CPU utilization caused by the web browser, and reactively respond to the changing values regardless of the type of workload or the performance requirements such as the target FPS. For web browsing, we define two types of workload, *foreground load* and *background load*. The workload caused by building up a web page until the user can interact with it is referred to as foreground load. After the page is built, background scripts are often executed and put a high background load on the system. The main difference between the two is, that the background load does not affect the user perception. Hence, it is not critical to process this kind of workload as fast as possible, what we exploit for power management. The interactive and ondemand governors, however, increase the CPU frequency as a response to the tasks' workload while it is unnecessary to finish them early. Such actions can significantly increase the power consumption and reduce the device's battery life. In addition, in many of the browsing activities, the web browser tries to maintain a target frame rate, e.g., 60 FPS. However, the Android governors are not aware of this, which can lead to slack times [9, 12, 13, 28].

In order to save power, while not sacrificing user experience, the coordination between an application and the underlying software components related to power management is crucial. Application-specific characteristics, rather than the CPU utilization alone, can give a better insight into the current resource demands of the application. In games, for example, there are different game phases such as *loading*, *menu*, and *playing* with varying workload and performance requirements [13]. During the loading phase, frames need not be updated as frequently as during the playing phase and the computation is memory bound. A similar observation holds true for web browsers as well. If the governor was aware of such contexts, or *phases*, it could reduce the CPU frequency without degrading the user experience. This scheme, as shown in Figure 1, is proposed as a basis for power management in this paper.

Illustrative example: Here, we describe a scenario where a click on a link within the Chrome browser loads a Reddit web page. The experiments were performed on an HMP platform with a Samsung Exynos5422 SoC, the Odroid-XU3 board [23] (see Section 4 for details). The board features a power-saving CPU (A7) and a performance-oriented CPU (A15). Figure 2 shows a time-wise plot of the power consumption, the usage, and the clock frequencies of the CPUs. It reveals that the power consumption of the A15 is considerably higher than the one of the A7, although the A7 usage is larger. The graphs on the left show the results for the ondemand governor, while the ones on the right show the results for our proposed phase-aware governor, which hereafter is referred to as the *browser governor*. While the ondemand governor regulates the CPU frequency based on the workload alone, we regulate the frequency based on the workload *and* the phase of the browser (see Section 6 for details). For the *Load* phase, both schemes behave similarly and the foreground load times are approximately similar (4.3 s), while the loading energy is slightly higher for the ondemand governor (7.7 J compared to 6.1 J for the browser governor). Even after the foreground load has completed, background load is still active for this web page. Here, our governor knows that the browser enters a *Load/Idle* phase that is not relevant for the user perception, as the page is already fully visible. Hence, it keeps the frequency of the high performance CPU (A15) at a low level and saves energy, unlike the ondemand governor. After the background tasks have finished, the web browser enters the *Idle* phase and the high performance cores can be powered down. The ondemand governor does not take this action, because powering up and down CPUs is accompanied by a time overhead. We address this problem by establishing a channel directly from the touch screen driver to the governor that is used to convey user input information. As a result, the energy consumption for the *Load/Idle* phase is reduced by 40 % from 8.2 J for the ondemand governor, to 4.9 J for our proposed browser governor.

Our contributions: In this paper – as illustrated through the previous example – we propose a *phase*-aware web browser power management scheme for HMP platforms, where the power manager in the underlying operating system is aware of the context that the web browser is in. Given the significant period of time we spend on web browsing on mobile devices, the complexity of today's web pages, and the impact they have on the smartphone's battery life, we believe that the changes we propose in the browser and the governor in the OS are fully justified. The resulting energy savings are significant, as will be discussed later in the paper. The main contributions of this work can be summarized as follows:

- We define web browsing *phases*, such as *Idle*, *Load*, *Scroll*, *Video*, etc., that exhibit distinct workload characteristics and user requirements, based on the internal information of the Chrome browser.
- We establish a channel between the application layer, the touch screen driver, and the governor, to directly share the phase information and react faster to events that trigger phase transitions.
- We implement a kernel governor – referred to as the browser governor – that controls the CPU power state and its voltage and frequency according to the available phase information.
- We demonstrate the effectiveness of this approach in terms of power consumption as well as responsiveness of the system.

It may be noted that such browser-driven (i.e., single application-driven) power management, as proposed in this paper, is acceptable in case of mobile devices, since unlike in desktops or laptops, they usually run one foreground application at a time. In other words, when a browser is being *used*, other applications are either sleeping in the background or closed. In case of systems that support the true use of multiple applications at the same time, we envision that each application would convey its *phase* to the OS, which would then take power management decisions that are compatible with all these applications. Such a generalization of the API that we propose in this paper might become relevant in future mobile devices.

The rest of the paper is organized as follows. Section 2 summarizes the related work in this domain. Section 3 provides background information about web browsers, and discusses the nature of internal information that is available. This is followed by Section 4 that describes the commonly used HMP platforms in modern smartphones and the default Android power management in such systems. Section 5 discusses different web browsing phases, followed by Section 6 that presents the corresponding power management strategies, and elaborates the overall architecture and the modifications to the OS kernel that we propose. We present our experimental results in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

Various aspects of web browsing and browsers have lately received considerable attention both in industry and in academia. A number of recent publications have targeted the performance of browsers [2, 7, 10, 22, 26] with the aim of improving the user experience. However, very few studies have addressed the issue of web browser power consumption, although battery lifetime is an important metric when measuring the usability of mobile devices.

There have been studies on managing power consumption by considering the wireless link [27, 44, 46]. The works [27] and [44] analyze the 3G protocol and suggest the reorganization of data transmission phases. Combining multiple fragmented transmissions into larger chunks gives more room for the wireless link to enter low power states and save power during web page loading. The work in [46] investigates the effect of transmission data rates on the CPU power consumption. It concludes that CPU idle time, which increases with lower data rates, has a significant impact on the power consumption, and data rates can be used as an indicator for DVFS. Our work is orthogonal to these techniques and will be able to provide additional power savings when used together with them.

Furthermore, power reduction techniques exploiting web page-specific characteristics have been explored. In [41], the impacts on power consumption of different web page components such as JavaScript, images and CSS have been analyzed. Here, multiple strategies have been proposed to save power, targeting web page re-organization and computation offloading. In [47, 48], a power management technique for big.LITTLE platforms was introduced, which chooses an appropriate CPU for a particular web page. It uses a predictive model that is trained using web page primitives such as CSS and HTML tags. Another approach profiles user and system events to identify the quality of service (QoS) required by a mobile web application [45]. This data is used to perform CPU task allocation and DVFS on a big.LITTLE platform. The main distinguishing feature of our work over the above works is that while they use *indirect* information of the context – e.g., web page primitives or user events – we make use of a browser's internal information *directly*, which allows more effective power management. For example, we use states such as *Video* or *Load/Idle*, which are not detectable from user events as in [45] (see Section 5).

Recent work has proposed to perform power management within the browser itself [3]. It analyzes the energy consumption for mobile web page loading and implements modifications to the browser to save energy on a big.LITTLE platform. The work also proposes to let the browser be aware of the underlying hardware, and directly handle thread scheduling. While this approach proved to be effective, this requires modifications in the browser that are specific to the hardware platform. In our opinion, the OS should handle hardware-specific operations for portability reasons, while the user space applications remain independent of the hardware. Our work proposes that the browser should only convey *phase* information to the OS such that the OS would perform better power management.

Further, *RECON*, a model of the energy consumption of mobile web page loading [4] and a detailed analysis of the impact of the underlying platform architecture on web browser power consumption [35] have been presented. In [35], experimental results from an HMP platform using various configurations such as varying CPU frequencies and CPU core configurations have been discussed. The conclusion is that by sacrificing browser performance marginally, a significant amount of power may be saved. However, no new power management strategy – i.e., no new governor – has been proposed in neither [4] nor [35], while in this paper we propose a *new* governor and compare it with the existing ones in Android. Our work extends the studies in [46], [35] and [4]. By taking cues from these results, we decided to design a new governor that directly exploits a browser's internal *phases* for power management.

Finally, the idea of establishing a communication channel between applications and the power manager was proposed in the past, but targeting different applications. For example, there has been work on mobile games [11, 13, 34], navigation and media streaming [14, 30]. All the works exploit the application-specific information provided by the respective application for power management. Another work proposes a more general approach that shares our idea of phases [8]. This work implements a programming language named *Energy Types* (ET), where energy phases are passed to the compiler and translated to power management strategies. To the best of our knowledge, this is the first proposal for a browser-specific Linux kernel governor for HMP platforms.

3 WEB BROWSING CHARACTERISTICS

In this section, we discuss the background information on web browsers that is necessary for understanding our work. First, we explain how web pages are represented within a browser. We then describe important implementation features of the browser that we have used for the purpose of this work – the Chrome browser.

3.1 Web Page Representation in a Browser

The web browser is a complex application that transforms a set of commands into the representation of the web page that we can see on the display and that we interact with. The page consists of static elements such as HTML and CSS that describe its layout and style. The dynamic behaviors of web pages, such as animations or user interactions, are mostly handled by JavaScript. The browser must guarantee a smooth interaction between the user and the web page.

Figure 3 shows how the browser creates a usable web page. The Document Object Model (DOM) tree is the internal representation of the page and is generated from the current web frame. The web frame is a snapshot of the static code and the dynamic modifications of this code by e.g., JavaScript. From the DOM tree, a layout tree is created, that contains information to display the web page elements – such as style rules. The paint layer tree combines the layout objects and groups them by the entities that will be displayed in the same coordinate space. The graphics layer contains already painted elements that are composited to a displayable web page and are rendered to the display by the GPU. One graphics layer can contain multiple paint layer trees. These data structures are created while the page is being loaded. Whenever anything changes, e.g., an animation is triggered by a script, the tree structure has to be updated, as shown in Figure 3. In general, the web browser targets a frame rate of 60 FPS, which is synchronized with the VSync signal of the display. During the computation of one frame, all of the above steps have to be completed before the next VSync signal is issued.

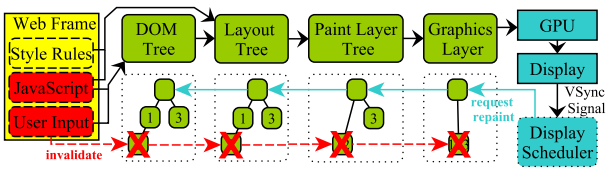

Figure 3: Updating a web page within the browser.

3.2 Browser Implementation Details

This section outlines the relevant implementation details of the Chrome browser. We intend to perform DVFS and power gating of the A15, hence, it is important to understand what sort of workload is generated by the browser. As mentioned, we have used the Chrome browser [20] for all of our experiments. The share of Chrome worldwide for mobiles and tablets is almost 60% [31]. The second largest share is hold by Safari (30%). Given these numbers, our implementation reaches most users of open-source browsers. However, it is generally possible to retrieve similar information from other browsers and apply our approach to them.

Processes: Chrome is divided into three processes, the *browser*, the *renderer* and the *gpu* process. As the names suggest, the browser process provides the user interface, the renderer process builds up the web page and the gpu process issues GPU commands to the display. All three processes maintain child threads. The renderer maintains a helper thread to manage web page contents: The *compositor* thread. The compositor holds a copy of the web page tree that was created by the main renderer to ensure the responsiveness of the browser. The main renderer can be blocked for different reasons, e.g., JavaScript or background script loading – both highly resource consuming actions. This would lead to delays and degrade the user perception. To overcome this, the compositor deals with user interactions, such as scrolling, in place of the main renderer.

Browser state information: There is a considerable amount of information in the previously described threads, which may be exploited by governors for the purpose of power management. For example, the browser tracks its own loading state. This can be used to distinguish between foreground and background load when loading a web page, which could be useful for reducing power without degrading user experience. The browser also maintains information about video streams, scrolling speed, etc. However, current browsers and also the Android system do not have any mechanism to communicate such information with each other. In Sections 5 and 6, we elaborate how this can be enabled and taken advantage of for the purpose of power management.

4 ANDROID POWER MANAGEMENT FOR HMP ARCHITECTURES

This section describes the specific HMP platform used for our experiments. We also describe how the Android default power manager works and discuss its deficiencies.

4.1 Android System Design

The structure of the Android OS that runs the Chrome browser is shown in Figure 4. The Android OS is divided in two parts, the application layer and the kernel. The kernel contains all the hardware drivers and is implemented partly in assembly and mostly in C. As in Linux, the kernel drivers can be accessed via the file system. The applications are usually designed in Java. However, for better performance there exists the so-called Java Native Interface (JNI), which allows parts of the application to be implemented in C++.

4.2 Odroid-XU3 HMP Platform

The platform that we use in this work is the Odroid-XU3 [23] development board. It features the Exynos5422 SoC, that is also

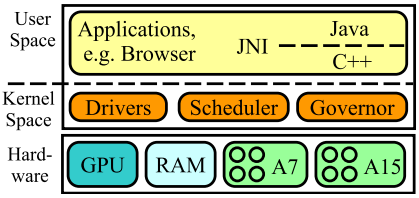

Figure 4: Android OS system structure.

built in the Samsung Galaxy S5 smartphone. The chip is based on the ARM big.LITTLE architecture with a power-saving *little* CPU, the A7, and a performance-oriented *big* CPU, the A15. Both CPUs contain four separate cores. In the following, we refer to core clusters such as A7 or A15 as *CPUs* and single CPU cores as *core*. The frequency levels of the CPUs can be controlled separately. The A7 can be operated from 1.0 GHz to 1.4 GHz while the A15 can be operated from 1.2 GHz to 2.0 GHz. Further, the platform supports power gating at CPU granularity, but due to Android limitations, only the A15 can be power gated during run time. The OS on this platform is an Android Kitkat 4.4.4 which is based on a Linux kernel version 3.10.9. The Chrome version that we run is 61.0.3139.0.

In addition, the Odroid-XU3 board contains four INA231 sensors that measure the power consumption of the A7, A15, the GPU and the RAM. These sensors can be read from the Android OS directly. We have instrumented the sensors for our measurements and measure the A7, A15 and GPU power values at approximately 1 kHz.

4.3 Power Management for HMP Platforms

The default Android power management system is divided in three separate entities: The task *scheduler*, the frequency *governor* and the *wakelock* mechanism. These entities work independently of each other. For each CPU, there is a governor and a scheduler. The scheduler decides which task is executed based on its priority. The governor monitors the workload of the cores and adjusts the frequency of the CPU depending on the workload. State-of-the-art governors, such as *ondemand* and *interactive*, tend to ramp up the frequency to the maximum while the workload is high. The wakelock keeps the CPU active as long as work needs to be done, e.g., if an application is currently running on the device. For HMP platforms, an additional HMP scheduler was designed, which decides whether a task is allocated to the A7 or the A15. It prefers the A7, but migrates a task to the A15 if the utilization of the task surpasses a certain threshold. The HMP scheduler only considers active CPU cores and does not migrate a task to power gated cores.

4.4 Android Power Management Limitations

The drawback of current Android power management is that it lacks communication channels among the power management entities in the kernel and the user applications. For example, a web browser cannot deliver the information about performance requirements to the underlying operating system to meet user QoS expectations. This results in over- or under-achieving the performance goals, e.g., frame rate or load time, and losing the potential for additional power reduction as we have seen in the previous illustrative example.

Another important issue is that Android default governors mainly use the lowest and the highest frequency levels, while many CPUs allow a variety of frequency levels. Considering the fact that the power consumption at high frequency levels is disproportionally higher than the associated performance gains, such a strategy is not beneficial for power savings. Table 1 shows the A15 frequencies and idle power consumptions corresponding to their voltage levels. While the increase in frequency from 1.2 GHz to 2.0 GHz would result in a maximum performance gain of 1.67, the power consumption increases by a factor of 3.38x. Further, the default governor's reluctance to power gate the A15 processor increases the system's power consumption. Many applications such as text messaging, timers, etc. can be run solely on the little CPU without any adverse performance impact. However, instead of putting the big CPU to sleep during such occasions, the A15 remains in an idle state in order to avoid the high wakeup delays. We have measured that a power-gated A15 consumes only about 0.045 W, while its idle power is 0.26 W at the lowest frequency level (a factor of 6x larger).

Table 1: Frequency, idle power and voltage of the A15.

Freq. [GHz]	Idle Pow. [W]	Voltage [V]
1.2	0.26	1.0
1.3	0.30	1.0
1.4	0.33	1.0
1.5	0.37	1.0
1.6	0.44	1.1
1.7	0.51	1.1
1.8	0.58	1.1
1.9	0.69	1.2
2.0	0.88	1.3

5 WEB BROWSING PHASES

In this section, we explain the browsing phases that we exploit for power management. First, we introduce the user-centric performance model RAIL in Chrome. Then, we define *phases* with different performance requirements based on the RAIL model.

5.1 User-Centric Performance Model RAIL

Within the Chrome browser, the user performance requirements are determined by the so-called user-centric *RAIL* model [21]. RAIL aims to provide a fast and smooth browsing experience. It defines the performance targets for the *Response*, *Animation*, *Idle* and *Load* (RAIL) phases as shown in Table 2, that have been adapted from the Human-Computer Interaction (HCI) domain [32]. Generally, there are two metrics to classify performance or QoS for web browsing: The response latency and the frame rate. The response latency is the time that the user needs to wait for an action to complete, e.g., for a web page to finish loading. The frame rate, usually measured in FPS, is used as a metric for animations such as scrolling and video play. For example, animations should be handled within 16.7 ms which means that the target frame rate is 60 FPS. However, the beginning of an animation may take up to 100 ms. It is also notable that the maximum web page loading time on mobiles is restricted to 5 s. This latency only refers to the loading time that the browser needs to make the page ready to use (foreground load). The background scripts, associated with advertisements etc., which may still be executed afterwards, are not bound by this constraint. This background load is not visible to the user and the total loading time of a page can be longer than the bound. RAIL is the desired behavior of Chrome,

Table 2: Summary of the RAIL model [21].

RAIL Step	Latency	(User) Actions
Response, Animation	< 16 ms	User drags finger and app's response is bound to finger position, ongoing page scroll/animation
Response, Animation	< 100 ms	User taps an icon/button, initiates page scroll, animation begins
Idle	–	Background activities
Load	< 5 s	Page ready to use on mobiles (foreground load only)

but the browser does not necessarily meet the target values defined by this model. For our power management strategy, we take a cue from the RAIL model to define corresponding browsing phases.

5.2 Definition of Browsing Phases

In this section, we introduce the phases that we have defined based on web browsing activities and the characteristics of the HMP architecture of our hardware platform.

5.2.1 Regular Browsing Phases. Naturally, web browsing consists of a sequence of different and repetitive actions. The RAIL model itself introduces phases such as response, animation, idle and load. We have defined our own phases (→ phase) based on the RAIL model and extended it where needed.

One of the most important browsing actions is loading a web page (→ phase *Load*). As already mentioned, (foreground) load is defined as the time it needs to build up the page until the user is able to interact with it. The browser provides a state value that indicates when this foreground load has finished. Background scripts may still be processed afterwards. This may generate high workload that is not critical for the user perception. Consequently, if the background scripts finish without any user interaction, this results in a temporary waiting state (→ phase *Load/Idle*). When all background actions have completed, the system enters a true idle state (→ phase *Idle*). Normally, there is interaction between the user and the web page, e.g., scrolling actions (→ phase *Scroll*). As a result of scrolling or even during idling, new scripts within the web page can be triggered, e.g., loading new Facebook posts. This can cause network traffic and, as a consequence, additional workload (→ phase *Load/Intermediate*). Further, the user can also trigger video play (→ phase *Video*). Both, *Scroll* and *Video* phases are derivatives of the RAIL mode animation.

5.2.2 Touch Events. One additional browsing phase results from the RAIL mode response, the (→ phase *Touch*). Responsiveness means that events triggered by the user are handled as fast as possible. As mentioned in the previous sections, this will pose a challenge if power gating the A15 shall be exploited to save energy, because the time overhead until the A15 cores are active again would hinder fast response to user input. This does not cause a problem for governors that do not power gate the A15. We introduce a workaround to ensure that the A15 is available on user interaction, because we assume that the workload will rise significantly after a touch event. Therefore, we detect touch events within the kernel and power up the A15 to prevent additional delays caused by power gating (see Section 6 for details).

6 PHASE-AWARE WEB BROWSER POWER MANAGER

Now that we have defined a number of phases, we propose an individual power management strategy for each of these phases. We also describe the implementation of the proposed browser governor within the Android OS.

6.1 Phase-aware Power Management Strategies

As depicted in Figure 5, our power management strategies are based on the workload and the user requirements in each phase. In our illustrative example, we have shown that the power consumption of the A15 is considerably higher than the power consumption of the A7. Hence, our main goal is to restrict the usage of the A15 and lower the power consumption by applying DVFS and power gating while maintaining a good user experience. We define the user requirements based on the RAIL model. In the following, we describe what type of power manager we have implemented in each phase and why. Note that when we refer to workload, we mean the workload on one core of a particular CPU. A high workload implies that one particular thread is the bottleneck of an application. Generally, we have implemented our DVFS strategy following the principles of the ondemand governor. This means that we increase the CPU frequency if the workload exceeds a given threshold. We have defined this threshold as 90 %, as used in the interactive governor, while it is 80 % for the ondemand governor. A lower load threshold will lead to an under-utilization of the CPU. Hence, all governors set a threshold of 80-90% and we follow the same practice. Moreover, we immediately decrease the CPU frequency when the workload is below the threshold. For brevity, we refer to this strategy as *performing DVFS* in the following. Additionally, our governor turns on and off the A15 by monitoring the phase and CPU workload, so that the default scheduler can migrate high workload tasks to the A15 when it is available. A transition graph that depicts when and why a phase change occurs is shown in Figure 6. Note that our governor performs a strategy comparable to the ondemand governor, when no browser workload is currently executing. However, this strategy can be easily adapted due to a modular source code design.

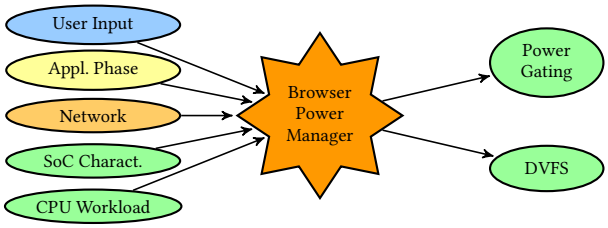

Figure 5: Browser governor information flow.

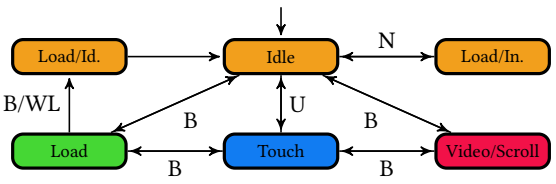

Figure 6: Phase state transition diagram of the browser governor. The transitions are based on user inputs (U), browser state changes (B), network traffic (N) or CPU workload (WL).

Idle: There is neither interaction from the user nor any network activity. We minimize the A7 frequency and turn off the A15.

Load/Intermediate: This is an idle state that deals with increased workload based on network activity. Increased network activity can be triggered by scrolling or animations. For example, scrolling down the Facebook page can trigger the download of new contents that need to be displayed, even when the scroll action is over. To deal with such scenarios, we turn on the A15 and allow the governor to perform DVFS for the A7 based on the workload.

Load: The load state is forwarded from the browser to the governor. We know that the load action is highly resource demanding. To guarantee the best user experience, we ramp the frequency of both CPUs up to the maximum when this phase is entered. Afterwards, the frequency of both CPUs is adjusted by performing DVFS.

Load/Idle: *Load/Idle* can only be entered when the browser reports that the actual *Load* phase is over. This phase becomes active if the workload remains high, although the load itself has finished. This may be due to background scripts. In this phase, we manage the frequency of the A7 by performing DVFS and fix the A15 frequency to its minimum possible value to save energy. By keeping the A15 active, we do not create a bottleneck in case the user starts interacting with the browser. The phase changes to *Idle* when the workload of both the A7 and the A15 falls below a minimum threshold of 200 % (25 % on each core). This value has been determined empirically and may be fine-tuned.

Touch: This phase was introduced to increase the responsiveness of the browser during phase transitions from *Idle* to interactive phases such as *Scroll* or *Load*. It is needed because powering up the A15 comes with a time penalty. Without the *Touch* phase, the browser governor would wait for the browser to process the touch event and calculate the next phase, e.g., *Load*. The regular touch propagation path in our setup is shown in Figure 7 on the left. It works for the default governors because they only need to ramp up the frequency based on the workload as explained in Section 4.3. However, the browser governor power gates the A15, which adds significant wakeup overhead to the touch event response time.

To solve the above issue, we power up the A15 at touch start and go back to *Idle* after a timeout of $\delta = 1.5$ s if no other phase change has occurred in the meantime. The same philosophy as used by the interactive governor is applied here: There is likely to be large workload and tight response time constraints after a touch event. We have determined this timeout parameter δ as follows. First, we have measured that the A15 needs on average of 9 ms for startup and 127 ms for power down. In the extreme case that the A15 needs to power back up immediately after a shutdown command was issued, there is a delay of 136 ms. Note that the timing overhead is the critical aspect why power gating is not practiced by the Android default governors. However, turning the A15 off and immediately back on again does not make a significant difference in energy consumption. We have defined the startup time as the time between calling the `cpu_up()` function for a CPU and its registration within the `cpufreq` module. Equivalently, we have defined the power down time as the time between a call to `cpu_down()` and its de-registration within the `cpufreq` module. Second, we have measured the load time for different $\delta = \{1.0, 1.5, 2.0\}$ s. While there was a significant performance degradation for $\delta = 1.0$ s, there was no performance improvement for $\delta = 2.0$ s compared to $\delta = 1.5$ s.

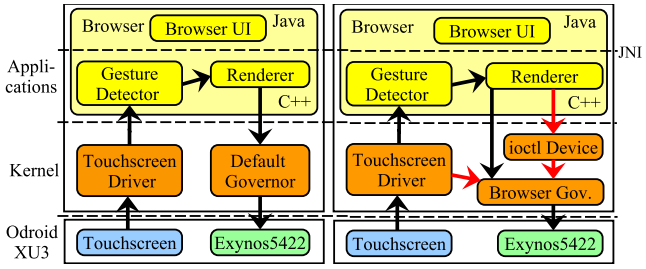

Figure 7: Regular touch event propagation (left) and browser governor (right) within the Android OS.

Scroll: The *Scroll* phase power management is based on the scroll speed and the frame rate. Both are passed from the browser to the governor. In general, we have observed that scrolling is not a costly operation. This is due to the division of the browser rendering engine into the main renderer thread and the compositor, as we have described in Section 3.2. As suggested by the RAIL model, we target an FPS value of 60.

The power management strategy is based on monitoring the frame rate and the workload. We have chosen to work with workload and frame rate *ranges* to avoid an oscillation of the CPU frequency. Hence, we are effectively targeting a value of 55 ± 5 FPS. If the workload is above 90% and the frame rate below 50, we increase the frequency. If the workload is below 80% and the frame rate above 55, we decrease the frequency. The A15 is turned on if the A7 cannot meet the FPS requirements by itself. Note that we usually enter the *Scroll* phase from the *Touch* phase. Hence, the A15 is initially turned on.

Video: We enter the *Video* phase based on browser information. As the browser does not provide the current video target frame rate, we target 30 FPS. This is a commonly used setting on video platforms such as YouTube. Otherwise, the power management strategy is the same as for the *Scroll* phase. We are aware that there exist videos with a higher FPS rate. However, we have postponed such detection strategies to future work.

6.2 Power Manager Implementation

In this section, we describe the implementation details of the proposed power manager. We explain the software changes that we made to the Linux kernel in the Android OS and the browser.

Kernel modifications: We have implemented the phase-aware power manager as a CPU frequency governor, an own module residing in the *cpufreq* domain of the Linux kernel. The system structure is shown in Figure 7 on the right. Within the governor, we expose a so-called *ioctl* device to the system, which can be accessed by the browser to pass information to the governor (frame rate, etc.). Such information is used to control the frequency of the CPUs, as well as the power state of the A15 within the implementation of the previously described power management strategies.

As already mentioned in the previous section, we have also implemented an additional kernel module that forwards user input information directly from the corresponding touch driver to our governor. This *shortcut* is depicted in Figure 7 on the right. One may note that such modules already exist in other Android systems [40]. Unfortunately, this was not the case in our platform at the time

of writing. Hence, we have implemented our own module that propagates the start of touch events directly from the touch screen driver to our governor. To minimize modifications of existing drivers, we have instrumented so-called kernel *notifier chains*. Using this method, any module (in our case the governor) can register itself to be notified whenever a particular event happens. The notification process can be triggered by any other module, for example by different touch screen drivers. We make use of this *shortcut* to alleviate the overhead caused by power gating the A15.

Browser modifications: In order to deliver phase information to the kernel governor, we also had to modify the Chrome browser appropriately. While the kernel governor provides the *ioctl* device itself, the browser passes data to the governor by writing to this device. The *ioctl* device can be accessed by standard file writing operations. The challenging part on the browser side was to actually find the right information within the browser source code. The information that we pass to the governor is the load state, frame rate, scrolling speed and video information.

7 EXPERIMENTAL RESULTS

In this section, we present our experimental setup and the resulting energy savings for the different web browsing phases. We show that a significant amount of energy can be saved – which would translate into a longer battery life – by exploiting the phase information of the browser for power management. Note that we have extracted the frame rate and the loading times for the measurements using the Chrome trace tool. The web pages were chosen partly from Alexa Top 50 web pages [1] and the Google Telemetry test suite [19].

7.1 Idle Phase

To measure the energy consumption during the *Idle* phase, we waited for the *Load/Idle* phase to complete and then measured the power consumed over a period of 10 seconds. The results obtained are shown in Figure 8. Each bar has three sections corresponding to the power consumed by A7 (bottom), A15 (middle) and the GPU (top). In Figure 8 and some of the next figures, the GPU power consumption is barely visible. As the system is idling, the results for different web pages are very similar. Some pages – Amazon, CNN and BBC – exhibit a slightly higher energy consumption. This is due to workload caused by animations. The maximum energy savings using our proposed browser governor are 57.4 % (CNN) compared to the interactive and 54.2 % (Amazon) compared to the ondemand governor. The mean savings are 52.0 % and 51.5 %, respectively. Figure 8 clearly shows, that the high savings result from the power down of the A15. These results emphasize the importance of the power gating strategies that we adopted. These strategies result in large energy savings and consequently increase the battery life time of the mobile device. Note that we would not be able to apply such aggressive power saving techniques if we were not aware of the current browsing phase. Therefore, these results clearly highlight the effect of sharing the phase information for power management.

7.2 Load Phase

Here, we measured the time and the energy from activating a link to a web page until this page has finished the foreground load. The link was activated by tapping on the screen with a finger. By designing

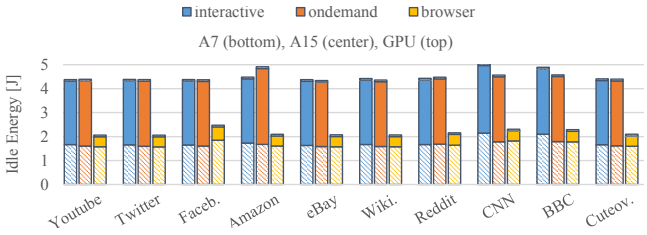

Figure 8: *Idle* phase energy consumption divided by consumers (A7, A15, GPU) for different governors.

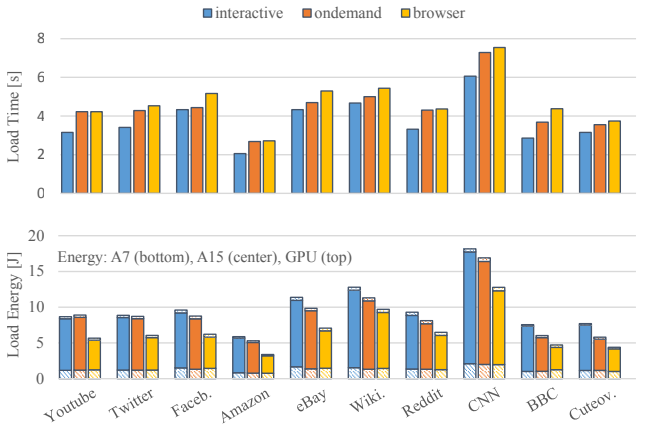

Figure 9: *Load* phase energy consumption (bottom) divided by consumers A7, A15 and GPU and load time (top).

the experiment in such a fashion, we also evaluate the effect of the *Touch* phase. Using our setup, we can extract the start time of the touch event directly from the kernel. The end of the foreground load is provided by the browser. The results are presented in Figure 9. The browser governor achieves significant energy savings for the *Load* phase – at maximum 36.3 % over the ondemand (YouTube) and 42.5 % over the interactive governor (Amazon). The mean savings achieved by our governor over the ondemand is 25.3 % and over the interactive governor is 33.4 %. On average, the loading time increased by 0.4 s (8.1 %) over the ondemand and by 1.1 s (28.2 %) over the interactive governor. There are two reasons for this: First, we perform a more aggressive DVFS strategy compared to the interactive governor. Second, the internal browser load state is activated considerably late. As a result, our *Load* phase power management technique becomes active later than the workload based techniques in the Android default governors. This problem can not be completely alleviated by the *Touch* phase. However, the *Touch* phase certainly reduces the loading time in our case and without it the delay would have been much longer. We consider this additional overhead acceptable since it is not always perceptible and the energy savings are considerable.

7.3 Load/Idle Phase

The *Load/Idle* phase marks the time that a web page needs to process potential background scripts after the foreground load has finished. This phase is derived solely from the CPU workload, and there is no explicit indicator from the browser. Hence, we analyzed the CPU

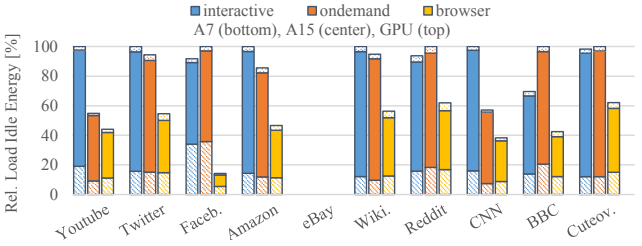

Figure 10: Normalized (see Section 7.3) background load energy consumption. The energy for eBay is zero.

workload to estimate the *Load/Idle* phase energy for the different governors. We defined the end of this phase as the time when the A15 workload has been zero for more than 2 s. The results are shown in Figure 10. As the absolute background load heavily varies across web pages, for each web page, we have normalized the energy consumption with respect to the governor for which the energy consumption is the maximum and plotted these normalized values. For example, for YouTube, the energy consumption with the interactive governor is the highest, and hence, it is at 100, while for BBC the ondemand is at 100. On average, the proposed browser governor saves 44.4 % and 50.5 % energy over the ondemand and the interactive governors, respectively. Note that some pages do not trigger any background scripts at all, e.g., eBay. As in the *Idle* phase, this test emphasizes the benefits of a phase-aware power manager. Again, we are able to demonstrate that there exists a large potential for energy savings with negligible impact on user-perceived QoS.

7.4 Video Phase

We played nine different videos from the YouTube platform for one minute each to evaluate the energy consumption of the video phase. We chose three videos showing *slowly* moving contents such as slide shows or barely moving contents. Further, we chose three videos with *medium* moving contents such as talk shows or animated movies. Last, we chose three videos containing *fast* action scenes. The energy consumption and the achieved frame rates are shown in Figure 11. As mentioned in Section 6.1, we target 30 FPS in this phase. The mean frame rate achieved by the interactive governor is 33.5, while the ondemand and the browser governors achieve 32.2 and 31.2 FPS, respectively. Although the frame rates across the different governors vary only slightly, the browser governor does a better job with power management as can be seen from the A15 power consumption (bottom plot in Figure 11). It saves up to a maximum of 26.4 % energy over ondemand and up to 35 % over the interactive governor among all the 9 evaluated videos. The mean savings applying the browser governor are 19.2 % over the ondemand and 29.0 % over the interactive governor. For fairness across achieved FPS, we provide the energy per frame value (top of Figure 11), which is constantly lower for our governor. This value expresses how much energy was spent on calculating one frame. The browser governor consumes 16.6 % less energy per frame than the ondemand and 23.6 % less than the interactive governor.

7.5 Scroll Phase

To evaluate the *Scroll* phase of the browser governor, we have recorded one long scroll gesture using the *reran* [16] tool and replayed it for all the web pages under test. As mentioned in Section 6.1, the *Touch* phase usually precedes the *Scroll* phase. This is not true when we simulate the gesture with reran, because reran does not trigger the touch driver. To work around this issue, we turned on the A15 before the test was performed. The total test duration was 2.6 s. As for the *Video* phase, we have measured the energy and the frame rate as performance indicators. The results are shown in Figure 12. The mean frame rate achieved by the interactive governor is 54.9, while the ondemand and the browser governors achieve 51.6 and 52.5 FPS, respectively. On average, the browser governor saves 25.1 % more energy over the interactive governor and consumes approximately the same energy (0.22 % more) as the ondemand governor. Our governor sometimes consumes more energy because it explicitly targets an FPS value between 50 and 60, while the ondemand governor is oblivious to FPS. However, our aggressive power management utilizing A15 power gating can lead to non-optimal FPS results, as for example for Google+. We plan to improve this in future work. Comparing the energy per frame values, the browser governor outperforms the interactive governor by 21 % and the ondemand governor by 1.7 % on average.

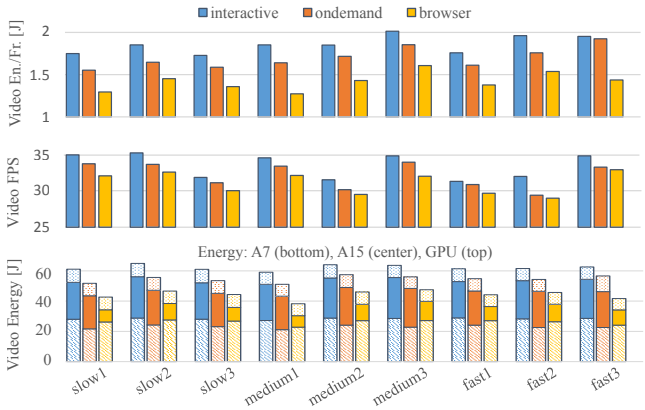

Figure 11: *Video* phase energy consumption over 60 s (bottom), frame rate (center), and energy per frame (top).

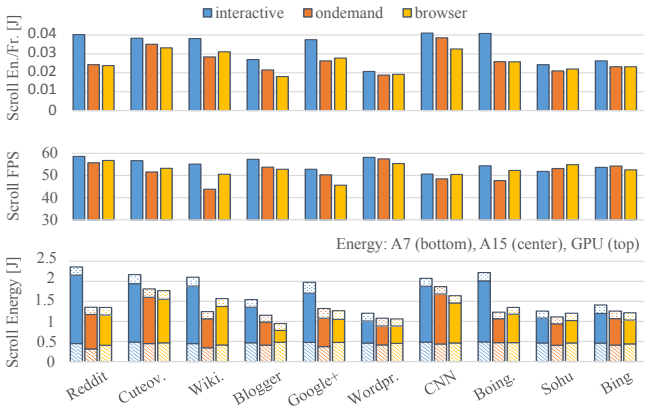

Figure 12: *Scroll* phase energy consumption (bottom), frame rate (center) and energy per frame (top).

While the browser governor sacrifices only 4.5 % performance (in FPS) compared to the interactive and performs even slightly better than the ondemand governor, the energy savings are significant. This shows that the browser governor performs well not only for different idle phases but also during interactive phases.

8 CONCLUDING REMARKS

In this paper, we introduced a phase-aware power manager for the Chrome browser. Towards this, we defined multiple *phases*, which differ in user performance requirements, and applied phase-specific power management strategies accordingly. We implemented a new governor that manages CPU frequencies and power states within an HMP platform, the Odroid-XU3 board, based on the information provided by the browser. We have shown that there exists a large potential for CPU energy savings when a browser's phase-specific characteristics are accounted for power management. In particular, up to 57.4% energy can be saved in idle phases, and 35 % in interactive/animation phases without noticeable degradation in user experience (albeit we did not conduct a user study, which is a future plan). The results also show that the performance overhead of our technique – mainly related to the power gating overhead associated with A15 – is manageable. As future work, we plan to generalize this power manager, such that it can be used by all Android applications.

Acknowledgments: This work was supported by Google Inc.

REFERENCES

[1] Alexa Internet, Inc. The top 500 sites on the web. http://www.alexa.com/topsites, 2016.
[2] Brave Software Inc. Brave. https://brave.com/, 2017.
[3] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking energyperformance trade-off in mobile web page loading. *GetMobile: Mobile Computing and Communications*, 20(2), 2016.
[4] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the energy consumption of the mobile page load. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1), 2017.
[5] M. Chan. cpufreq: interactive: New 'interactive' governor. https://lwn.net/Articles/662209/, 2015.
[6] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015.
[7] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. *SIGPLAN Not.*, 50(11), 2015.
[8] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.
[9] S. K. Datta, C. Bonnet, and N. Nikaein. Android power management: Current and future trends. In *The First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT)*, 2012.
[10] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. Idle time garbage collection scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.
[11] B. Dietrich and S. Chakraborty. Managing power for closed-source android os games by lightweight graphics instrumentation. In *Annual Workshop on Network and Systems Support for Games (NetGames)*, Nov 2012.
[12] B. Dietrich and S. Chakraborty. Forget the battery, let's play games! In *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2014.
[13] B. Dietrich and S. Chakraborty. Lightweight graphics instrumentation for game state-specific power management in Android. *Multimedia Systems*, 20(5), 2014.
[14] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1999.
[15] T. Garsiel. How browsers work. http://taligarsiel.com/Projects/howbrowserswork1.htm, 2009.
[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering (ICSE)*, 2013.
[17] Google Inc. Nexus 5X. https://www.google.com/nexus/5x/, 2015.
[18] Google Inc. There's an app for that...the browser, 2015.
[19] Google Inc. Catapult. https://chromium.googlesource.com/catapult/, 2017.
[20] Google Inc. Chrome. https://www.google.com/intl/en/chrome/browser/desktop/index.html, 2017.
[21] Google Inc. The RAIL Performance Model. https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail, 2018.
[22] S. K. Gudla, J. K. Sahoo, A. Singh, J. Bose, and N. Ahamed. Framework to improve the web application launch time. In *IEEE International Conference on Mobile Services (MS)*, 2016.
[23] Hardkernel co., Ltd. Odroid-XU3. http://www.hardkernel.com, 2015.
[24] Y. Huang, S. Chakraborty, and Y. Wang. Using offline bitstream analysis for power-aware video decoding in portable devices. In *ACM International Conference on Multimedia (MM)*, 2005.
[25] E. Kim, Y. Ko, and S. Ha. An adaptive frames per second-based CPU-GPU cooperative dynamic voltage and frequency scaling governing technique for mobile games. *J. Low Power Electronics*, 12(4):309–322, 2016.
[26] A. Knox and P. Seeling. Mobile web page characteristics: Delivery and stability considerations. In *IEEE Consumer Communications Networking Conference (CCNC)*, 2017.
[27] P. Kulkarni and P. Jaini. Android phone performance enhancement by energy efficient web browser. In *Global Conference on Communication Technologies (GCCT)*, 2015.
[28] W. Y. Liang and P. T. Lai. Design and implementation of a critical speed-based dvfs mechanism for the android operating system. In *International Conference on Embedded and Multimedia Computing*, 2010.
[29] M. Meeker. KPCB Internet Trends. http://www.kpcb.com/blog/2015-internet-trends, 2015.
[30] M. Martins and R. Fonseca. Application modes: A narrow interface for end-user power management in mobile devices. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2013.
[31] NetMarketShare. Browser Market Share. https://netmarketshare.com, 2018.
[32] J. Nielsen. *Usability Engineerings*. Morgan Kaufmann, 1993.
[33] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Linux Symposium*, 2006.
[34] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Annual Design Automation Conference (DAC)*, 2015.
[35] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford. Web browser workload characterization for power management on hmp platforms. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016.
[36] A. Pilon. Smartphone battery survey: Battery life considered important. https://aytm.com/blogmarket-pulse-research/smartphone-battery-survey/, 2016.
[37] Qualcomm Technologies, Inc. Qualcomm® Snapdragon™ 808. https://www.qualcomm.com/products/snapdragon/processors/808, 2015.
[38] B. Raman and S. Chakraborty. Application-specific workload shaping in multimedia-enabled personal mobile devices. *ACM Trans. Embedded Comput. Syst.*, 7(2):10:1–10:22, 2008.
[39] Samsung Electronics Co., Ltd. Mobile Processor Exynos 9 Series (8895). http://www.samsung.com/semiconductor/minisite/Exynos/Solution/MobileProcessor/Exynos_9_Series_8895.html, 2017.
[40] The Linux Foundation. CPU boost.
[41] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *21st International Conference on World Wide Web (WWW)*, 2012.
[42] A. Wei. 10 nm process rollout marching right along. http://www.techinsights.com/about-techinsights/overview/blog/10nm-rollout-marching-right-along/, 2017.
[43] J. Yu, H. Han, H. Zhu, Y. Chen, J. Yang, Y. Zhu, G. Xue, and M. Li. Sensing human-screen interaction for energy-efficient frame rate adaptation on smartphones. *IEEE Transactions on Mobile Computing*, 14(8), 2015.
[44] B. Zhao, W. Hu, Q. Zheng, and G. Cao. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems*, 26(3), 2015.
[45] Y. Zhu, M. Halpern, and V. Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.
[46] Y. Zhu, M. Halpern, and V. J. Reddi. The role of the cpu in energy-efficient mobile web browsing. *IEEE Micro*, 35(1), 2015.
[47] Y. Zhu and V. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
[48] Y. Zhu, A. Srikanth, J. Leng, and V. J. Reddi. Exploiting webpage characteristics for energy-efficient mobile web browsing. *IEEE Computer Architecture Letters*, 13(1), 2014.