TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Application-Aware Power Management for Interactive Android Applications on HMP Platforms

Nadja Heitmann

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. sc. techn. Andreas Herkersdorf
Prüfer der Dissertation:	1. Prof. Dr. sc. Samarjit Chakraborty
	2. Prof. Dr. Ir. Jeroen Voeten, Eindhoven University of Technology, Netherlands
	3. Priv.-Doz. Dr. Daniel Müller-Gritschneder (nur mündliche Prüfung)

Die Dissertation wurde am 11.02.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 07.10.2019 angenommen.

Most of this work was funded by a grant from Google, which we gratefully acknowledge.

Abstract

Thanks to the performance improvements in hardware and software architectures, more applications, which used to run only on desktop computers, are now being migrated to mobile devices. Nowadays, people spend even more time on mobile devices than on desktop computers. However, this entails increased power consumption, that necessitates more effective runtime power management techniques due to battery capacity constraints. Such techniques should reduce power consumption while satisfying user-perceived requirements in interactive applications, such as frame rate, and response times. A major hurdle in incorporating such techniques into real products is that user-perceived requirements are only visible to user applications, but not accessible by the power managers residing in the operating system. Software architectures that have worked well on desktops might not necessarily be optimal for the mobile counter parts, exemplified by the current power management software on Android systems. In this work, we show that better power management is achievable by passing such information to the operating system, and propose an Application Programming Interface (API) for that purpose. Therefore, we study two highly interactive applications to give insight into how such an API could be designed: Mobile games and web browsing.

Gaming workloads exhibit highly variable and user-interactive behavior, which makes it hard to predict the workload. Modern Multi-Processing System on Chips (MPSoCs) platforms are equipped with Heterogeneous Multi-Processing (HMP) processors comprising performance-oriented and energy-efficiency cores in order to better exploit power-performance trade-offs among different types of applications. To minimize the energy consumption of games on HMP platforms, it is essential to precisely predict the workload and perform joint thread-to-core allocation as well as Dynamic Voltage and Frequency Scaling (DVFS). We propose a frame- and thread-based MPSoC power management strategy for games. In this work, we focus on the fact that gaming workload has high temporal correlation between frames and evaluate selected workload predictors on a per-frame basis. Moreover, we find that there are two categories of thread workloads, periodic and aperiodic, and hence, propose to use a hybrid workload predictor. Based on per-thread predictions, the power manager allocates the threads among the heterogeneous cores in an evenly distributed fashion to minimize the operating frequency while keeping the Frames-per-Second (FPS) constraint. We implement the game power manager as an Android governor on a state-of-the-art ARM big.LITTLE HMP platform based on the Exynos5422 System on Chip (SoC), which is also incorporated in the Samsung Galaxy S5 smartphone. Our results show that we save on average 41.9 % of energy compared to the Android default governor. Further, we have performed a user study to evaluate the user perception of our governor. The gaming experience was rated between *good* and *very good* for all games.

Besides games, web browsers exhibit a variety of possible interactions with the user, such as loading a web page, scrolling or watching a video. The volume of mobile web browsing traffic has significantly increased as well as the complexity of the mobile web pages mandating high-performance web page rendering engines to be used on mobile devices. Although there has been a significant improvement in performance of web page rendering on mobile phones in recent years, the power consumption reduction has not been addressed much. A main contribution of this work is a thread level analysis of the workload generated by Google's Chrome browser on our HMP evaluation platform. We analyze the detailed traces of the thread workload generated by the web browser, especially the rendering engine, and discuss the power saving potentials in relation to power management policies in Android. Moreover, we performed a power versus performance analysis of the A15 Central Processing Unit (CPU) on our platform. The focus of our study lies on frequency capping, thread-to-core allocation and power gating and its effects on the loading time of web pages. Our work shows that large power savings come with small performance deficits. However, the amount of power that can be saved is significantly larger than, for example, the decrease in loading time. In particular, power gating the A15 reduces the idle power of the A15 by 85 %.

The findings of our browser power characterization motivated us to develop our own browser-aware power manager that takes into account the application's state, which mainly depends on the current user interaction. 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. Hence, 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 the HMP platform mentioned above. We believe that our work will lead to development of practical power management techniques for interactive applications considering collaborative thread-to-core allocation, DVFS and power gating as well as the communication between power managers and applications.

Zusammenfassung

In den vergangenen Jahren ist die Leistung von Hardware- und Softwarearchitekturen enorm gestiegen. Als Konsequenz werden immer mehr Applikationen, die ursprünglich für stationäre Computer entwickelt wurden, auf mobile Geräte portiert. Die Verbraucher benutzen heutzutage deutlich häufiger mobile Geräte als stationäre Computer. Dieses zieht einen höheren Energieverbrauch auf dem entsprechenden Gerät nach sich. Das macht es wiederum notwendig, effizientere Strategien zum Energiemanagement zu entwickeln, da die Batteriekapazität eines mobilen Geräts begrenzt ist. Solche Strategien sollten den Energieverbrauch reduzieren und gleichzeitig keinen Einfluss auf die Nutzerwahrnehmung haben. In interaktiven Applikationen beeinflussen primär die Bildwiederholungsrate und veränderte Ladezeiten der Applikation die Wahrnehmung durch den Benutzer. Eines der größten Hindernisse, um effizientere Strategien für das Energiemanagement in echte Produkte zu integrieren, ist, dass dem Betriebssystem, welches das Energiemanagement betreibt, keine Informationen über die Benutzeranforderungen von interaktiven Applikationen zur Verfügung stehen. Diese sind nur innerhalb der Applikation selbst verfügbar. Konsequenterweise sind Softwarearchitekturen, welche sich auf stationären Computern bewährt haben – gerade im Hinblick auf das Energiemanagement von Androidsystemen – nicht optimal auf mobile Systeme übertragbar. In dieser Arbeit legen wir ein API-Konzept dar, welches vorsieht, dass Benutzeranforderungen zwischen der Applikation und dem Betriebssystem geteilt werden, da dieses besseres Energiemanagement ermöglicht. Im Rahmen dessen stellen wir zwei in hohem Maße interaktive Applikationen vor, die Aufschluss darüber geben, wie so eine API gestaltet werden kann: Mobile Spiele und Webbrowser.

Die Arbeitslast des Prozessors bei Spielen ist sehr variabel und hängt von der Aktivität des Benutzers ab, was eine Vorhersage dieser Last schwierig macht. Moderne MPSoC-Plattformen sind mit HMP Prozessoren ausgestattet, welche leistungsorientierte und energieeffiziente CPU-Kerne vereinen. Das soll den variierenden Leistungsanforderungen und dem damit verbundenen Energieverbrauch von verschiedenen Applikationen entgegen kommen. Um den Energieverbrauch auf diesen Plattformen zu minimieren, ist es wichtig, die Arbeitslast von Spielen möglichst genau vorherzusagen. Basierend auf der Vorhersage können Threads bestimmten Kernen der CPU zugewiesen und DVFS (Regelung der CPU-Taktfrequenz) betrieben werden. Wir stellen hier eine Strategie zum Energiemanagement für Spiele vor, die auf der Bildwiederholungsrate und der Arbeitslast von einzelnen Threads basiert. Hierbei nutzen wir aus, dass zeitlich benachbarte Bilder eine korrelierende Arbeitslast aufweisen und evaluieren mehrere Last-Prädiktoren, die bildratenbasiert arbeiten. Außerdem zeigen wir, dass Spielethreads zwei verschiedene Kategorien von Arbeitslast aufweisen, periodische und aperiodische, und implementieren einen darauf basierten hybriden Prädiktor, der die Arbeitslast pro Thread pro Bild

voraussagt. Darauf basierend verteilt der Energiemanager die Threads auf den Kernen der HMP-Plattform. Hierbei versucht er, die Arbeitslast möglichst gleichmäßig zu verteilen, um die Taktfrequenz möglichst niedrig halten zu können, dabei aber die Anforderungen an die Bildwiederholungsrate zu erfüllen. Wir haben den Energiemanager als sogenannten *Android Governor* implementiert. Die Experimente werden auf einer modernen ARM big.LITTE HMP Plattform ausgeführt, die den Exynos5422 Chip verbaut hat. Dieser Chip wurde auch in dem Samsung Galaxy S5 Smartphone eingesetzt. Unsere Messergebnisse weisen einen durchschnittlich 41.9 % niedrigeren Energieverbrauch auf als die Standardgovernor in Android. Des Weiteren haben wir eine Studie durchgeführt, um die Nutzerwahrnehmung unseres Governors zu bewerten. Die Spielqualität wurde durchgängig als gut und sehr gut bewertet.

Auch der Webbrowser weist eine ganze Reihe von Interaktionen mit dem Benutzer auf, zum Beispiel das Laden einer Website, Scrollen oder das Gucken eines Videos. Da mobile Webseiten immer komplexer werden, hat das Datenvolumen im Zusammenhang mit dem mobilen Surfen signifikant zugenommen. Um diese korrekt darstellen zu können, werden performante *Rendering-Engines* benötigt, auch auf mobilen Endgeräten. Obwohl ihre Performanz in den letzten Jahren massiv zugenommen hat, wurde wenig auf den Energieverbrauch dieser Rendering-Engines Wert gelegt. Einen signifikanten Beitrag dieser Thesis stellt eine Lastanalyse der Threads dar, die vom Googles Chrome Browser erzeugt werden. Diese Analyse wird auf der bereits erwähnten HMP Plattform durchgeführt. Wir haben den genauen Lastverlauf der Threads analysiert, insbesondere den des Rendererthreads. Basierend auf dieser Analyse, zeigen wir Wege zum Einsparen von Energie in Androidsystemen auf. Außerdem haben wir eine Gegenüberstellung von Performanz und Energieverbrauch für die A15 CPU angefertigt. Konkret untersuchen wir den Effekt von Begrenzung der CPU-Taktfrequenz, der Zuordnung von Threads zu bestimmten CPU-Kernen und dem Ausschalten der A15 CPU auf die Ladezeit von Webseiten. Obwohl große Energieeinsparungen kleinere Performanzeinbußen mit sich bringen, sind die Einsparungen signifikant größer als beispielsweise die Zunahme der Ladezeit. Besonders das Ausschalten der A15 CPU reduziert die Leerlaufleistung der A15 CPU um 85 %.

Unsere Ergebnisse bezüglich der Energieanalyse des Browsers haben uns dazu motiviert, unseren eigenen Energiemanager zu entwickeln. Dieser berücksichtigt den jeweiligen Zustand des Browsers, welcher hauptsächlich von der Interaktion mit dem Benutzer abhängt. Browser bestehen aus vielen lastintensiven Komponenten wie der Rendering-Engine oder der JavaScript-Engine. Diese Komponenten generieren eine hohe Last ohne die Kapazitäten oder den Energieverbrauch der Hardware-Plattform in Betracht zu ziehen. Zusätzlich führt der nicht vorhandene Informationsaustausch zwischen der Browser-Applikation und dem Energiemanager im Betriebssystem (hier Android) zu einem hohen Energieverbrauch. Daher haben wir einen Energiemanager implementiert, welcher den internen Status des Browsers berücksichtigt – die sogenannte *Phase*. Am Beispiel des Google Chrome Browsers zeigen wir, dass unser Energiemanager 57 % Einsparung gegenüber den Standardmanagern von Android erreichen kann. Die Implementierung und Evaluierung unseres Governors erfolgte auf der oben erwähnten HMP-Plattform. Wir denken, dass unsere Arbeit ein Wegweiser für die Entwicklung von Energiemanagern sein könnte, welche auch in der Praxis eine Verbindung von Thread-CPU-Zuordnung, DVFS und dem gezielten Ausschalten von CPUs sowie die Kommunikation zwischen Applikationen und Energiemanagern in Betracht ziehen.

Contents

1 Introduction

Mobile devices have become an integral part of our daily lives. More and more applications are shifting from desktop computers to mobile devices such as smartphones and tablets. As a result, we spend more time on mobile devices than we do on desktop computers, nowadays [98]. This development is enabled by the fast enhancement of mobile hardware platforms that become more powerful with every new generation. The downside is a significant increase in the power consumption of the devices, which has made the battery runtime a major decision criteria for buying a smartphone [127]. As shown in Figure 1.1, long battery life has been the top decision criteria for buying a smartphone in Germany in 2014 and 2015. Hence, power management techniques for mobile devices have become very important in today's system design. This is especially true for interactive applications that produce performance peaks, which are highly resource demanding.

While the battery life of mobile devices is such an important issue for the user, power management is often not considered during the development of mobile applications – so-called *apps*. This is due to multiple reasons:

First, many applications have originated in the desktop world and have been ported to the mobile domain with the emergence of smartphones and tablets. As power consumption is not a critical issue for desktop computers, the focus of such applications is usually on performance. 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.

Second, the major drawback of the current Android software architecture is that the power management is done without regards to application-specific characteristics, which are not visible to the kernel. A key characteristic of applications running on mobile devices is that they are user experience-sensitive. For example, an application is expected to respond immediately to a touch event like scrolling or zooming, and maintain a certain frame rate during animations. On

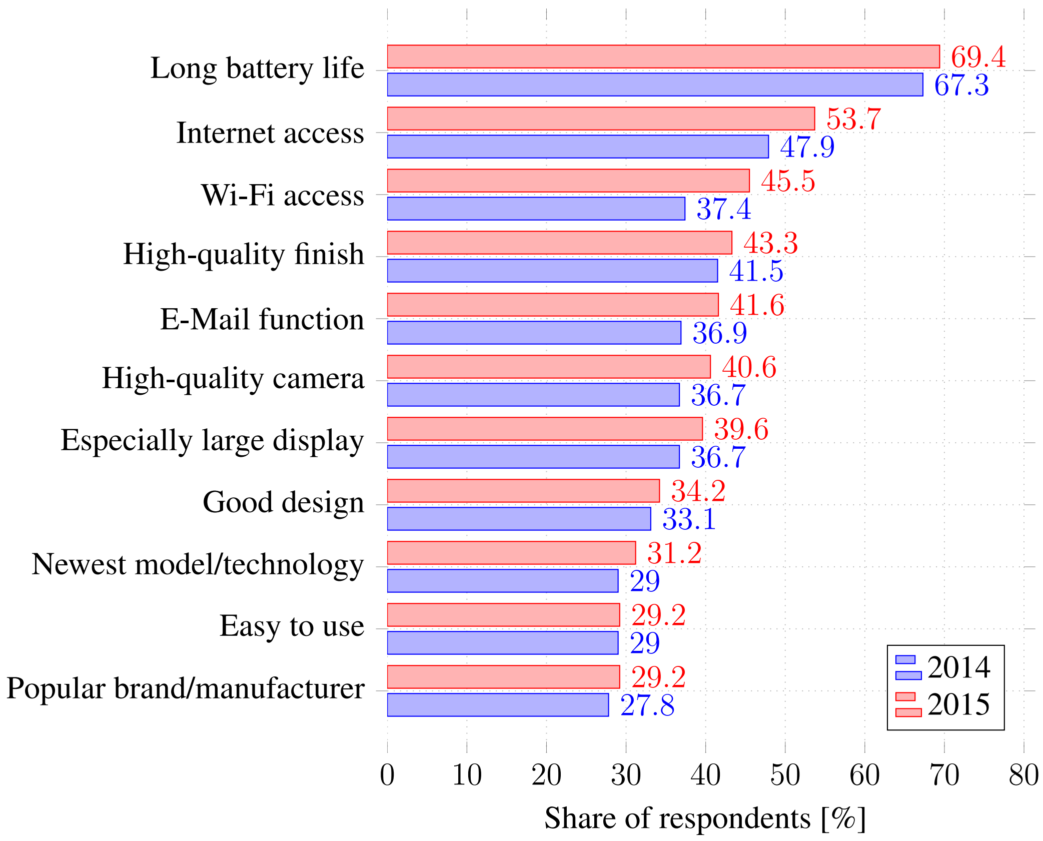


Figure 1.1: Most important criteria for buying a new mobile phone or smartphone in Germany from 2014 to 2015 [71] (excerpt).

the other hand, there are background tasks that have little impact on user experience. As there is no communication channel between the application and the power manager, the power management techniques in Android are purely workload-based and cannot consider application-specific information [15, 113].

Third, from the application programmers' point of view, the power management has been considered a job to be handled by the Operating System (OS) or at the hardware layer, and energy-aware software development has not yet been in the main focus of the community [128]. Again, this works well for devices without battery constraints, but causes problems for devices with limited battery runtime. Generally speaking, such a system design that separates kernel and application layer has been developed to ease the job of the programmer. It should let him or her focus on the functionality of the app without the necessity to consider other issues. As a result, there is a lack of awareness and a lack of knowledge in how to design power-aware applications. Another important issue is the lack of tools that help developers analyze their code and identify pitfalls that lead to higher power consumption.

The above shortcomings become even more distinctive when it comes to HMP platforms incorporating the ARM big.LITTLE architecture [92]. This architecture is adopted in state-of-the-art smartphones like the Nexus 5X with its Qualcomm Snapdragon 808 processor [43, 131], the Samsung Galaxy S8 with its Exynos Octa 8895 [140, 141], the Samsung Galaxy S9 with its

Exynos Octa 9810 [142, 143], the Huawei Mate 10 with its Kirin 970 [68] or the LG V40 ThinQ with its Qualcomm Snapdragon 845 [85, 132]. This particular SoC design combines a power-efficient (little) CPU which is less performant and a performance-oriented (big) CPU. Due to a larger scope of performance settings, which is described in Section 1.1, the power dissipation for such platforms is particularly high for workload-intensive applications. The goal of this work is to demonstrate better power management strategies for Android applications, especially on HMP platforms, by creating a connection between the application and the kernel, such that the kernel can incorporate application-specific information for its power management decisions.

The rest of this chapter is organized as follows: We first describe state-of-the-art HMP platforms with the main focus on the Odroid-XU3 [59] board that we have used for all of the experiments presented in this work. Then, we explain why the CPU power management, as currently performed on Android, does not achieve optimal results in terms of power consumption for such platforms. After that, we briefly outline our implemented power management techniques for the games and web browser applications. Finally, we summarize the contributions of this thesis and outline the organization of the remainder of this work.

1.1 Android Power Management for HMP Architectures

This section gives some background information of current mobile hardware architectures and the Android OS. As a representative example for a state-of-the-art SoC, we introduce the Odroid-XU3 board used in this work. We explain how the Android operating system power management works for this kind of architectures and where there exists possible improvement potential of the current Android power management strategy in terms of power consumption.

1.1.1 Android System Design

The structure of the Android OS that runs the different user applications is shown in Figure 1.2. Generally speaking, the Android OS is divided in two parts, the application layer and the kernel layer. The kernel contains all the hardware drivers and is implemented partly in assembly and mostly in C, highly depending on the underlying hardware platform that the system is running on. Similar to Linux, the kernel drivers can be accessed via the file system if an application owns the necessary permissions. The permissions can be defined by configuring the *ueventd.rc* file inside the Android file system. The applications are usually designed in Java. However, nowadays, the programming language Kotlin [75] can be used for application development as well. Moreover, for better performance there exists the so-called Java Native Interface (JNI), which allows parts of the application to be implemented in C++.

1.1.2 Odroid-XU3 HMP Platform

The platform that we use in this work is the Odroid-XU3 development board [59]. It features the Exynos5422 SoC, which is also 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

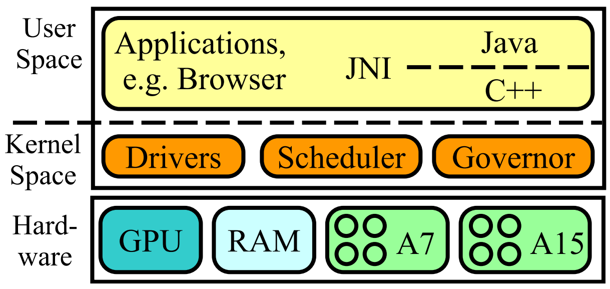


Figure 1.2: Android OS system structure.

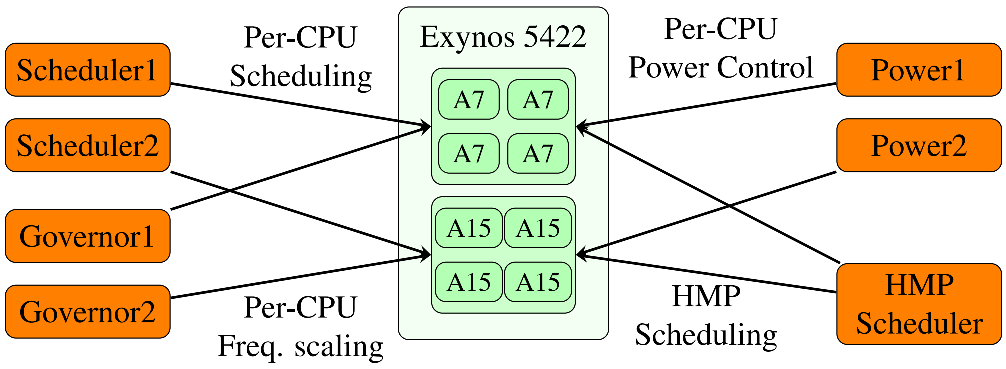


Figure 1.3: Traditional Android power management with independent control units.

following, we refer to CPU 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 (in 100 MHz steps). While there exist big.LITTLE derivations where only one CPU can be operated at a time, our platform is a so-called HMP platform. This implies that all eight CPU cores can be operated in parallel. 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.

1.1.3 Android Power Management

The default Android power management system is divided in three separate entities: The task *scheduler*, the frequency *governor* and the *wakelock* mechanism. The scheduler decides which tasks runs on which core and when. The governor controls the CPU frequency and the wakelock mechanism keeps hardware components in a power-up state while they are in use. The three of them work independently of each other as depicted in Figure 1.3. To understand why the Linux kernel is structured in this manner, it is worth looking at the development and enhancements of CPUs over the past years.

At the very beginning, the CPU was a single-core unit running at a fixed frequency. The only necessary control unit is the scheduler that decides which task is supposed to run at which time based on its priority. This simple hardware structure was soon enhanced by multi-core

CPUs that could run multiple tasks in parallel, an enhancement that requires a more involved scheduling technique. Moreover, CPUs were soon able to switch between different CPU frequencies, which requires a mechanism to control the frequency level. However, the number of frequencies was not as high as nowadays (e.g., nine different frequency levels for the A15 of the Odroid-XU3). The Android default CPU frequency governors monitor the workload of the cores and adjust the frequency of the CPU depending on the workload. State-of-the-art governors, such as the *ondemand* [113] and *interactive* [15], tend to simply ramp up the frequency to the maximum while the workload is high. Compared to the ondemand governor, the interactive governor was developed specifically for mobile devices and designed to react to user inputs fast. For simple architectures, it might be sufficient to switch to the highest CPU frequency while the workload is high. However, there is more flexibility for power management once the hardware architecture becomes more complex – not only to exploit more frequency levels but also to switch between CPUs on a big.LITTLE platform. For big.LITTLE platforms, there is a governor and a scheduler for each CPU. Note that different CPUs can have different governors.

For HMP platforms, an additional HMP scheduler was designed, which decides whether a task is allocated to the A7 or the A15 CPU. 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. The platform also supports CPU power gating, but due to performance reasons, this method is not applied by the Android power managers while the user is interacting with the device. Please note once again that a big.LITTLE architecture is not automatically an HMP architecture. There also exist older architectures where the system switches between either the big or the little cores.

1.1.4 Android Power Management Limitations

The division described above works well for traditional hardware architectures with only one CPU, but not for heterogeneous MPSoCs as used in this work. While wakelock mechanism and cpufreq work for both CPUs independently, the scheduler has to distribute the tasks over all available cores. Therefore, the HMP scheduler was designed by Samsung [25]. As mentioned above, it monitors the individual load of each process. When the load surpasses an upper threshold, the process is migrated to a big core and vice versa. At the same time, the CPU governor which works independently from the scheduler rises the frequency because of the increasing load. This causes maximum power consumption for all CPU intensive applications, in particular interactive ones. An example for this situation is shown in Figure 1.4. It shows the loading of the eBay web page using the Chrome browser and the regular Android settings on the Odroid-XU3 platform.

From this figure we can draw multiple important observations about the Android power management: (1) Although the workload on the A15 is clearly below 100% and the CPU is not fully utilized during the first two seconds, the frequency is at the maximum value of 2 GHz. Hence, DVFS could have been applied to reduce the frequency and generate a higher workload on the CPU. (2) The loading (or rendering) of the web page generates a high workload, and consequently, some work has been migrated to the A15. Although neither of the CPUs are fully utilized, one can see that the corresponding clock frequencies are switched between the mini-

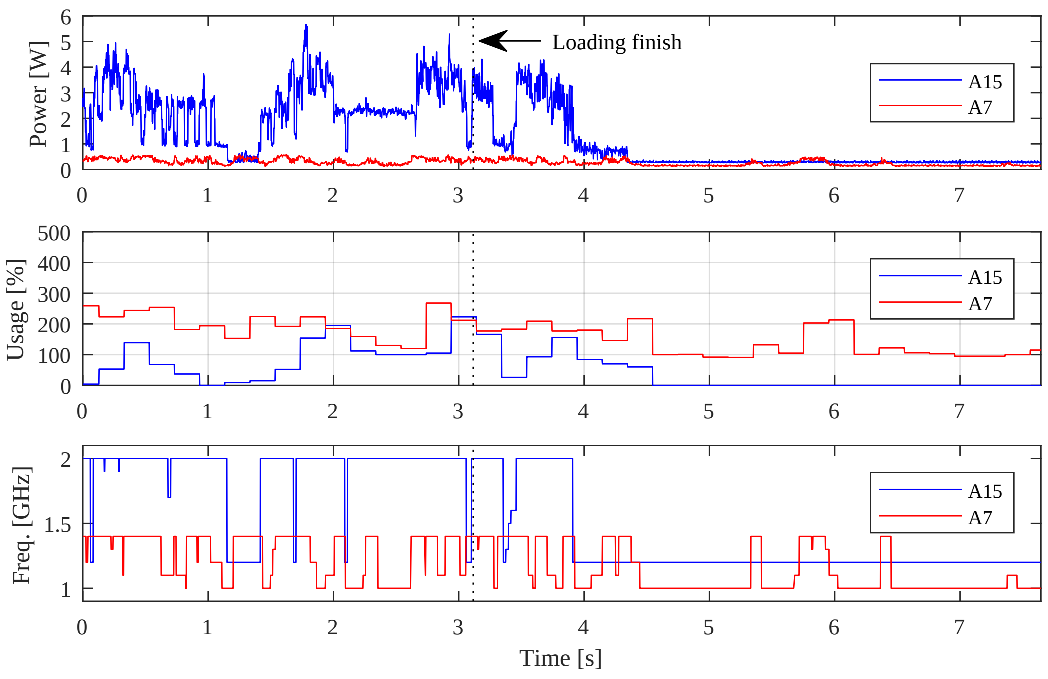


Figure 1.4: Loading eBay web page (regular Android settings).

mum and maximum values. More important, even when the CPU utilization is less than 100%, which means that even a single core is not fully utilized, the CPU is at its maximum frequency. The DVFS granularity for both CPUs is at 0.1 GHz though. Consequently, there is potential for power versus performance optimization by instrumenting the intermediate frequency levels. Considering the fact that the power consumption at high frequency levels is disproportionally higher than the associated performance gains, the default Android DVFS strategy is not beneficial for optimization towards power consumption. Table 1.1 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. (3) After loading the web page has finished, the utilization of the A15 drops to zero and its frequency to the lowest value of 1.2 GHz. Still, the idle power of the A15 is at the same level as the power consumption of the A7 although the A7 is still in use and the A15 is not. So, even when not in use, the A15 consumes about 50% of the total CPU power. At this point, the A15 could be put into a deeper power-saving state or be rather power gated. 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).

Another important issue in terms of efficient power management is that the Android system 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 Quality of Service (QoS) expec-

Table 1.1: 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

tations. This results in over- or under-achieving the performance goals, e.g., frame rate, load time, and losing the potential for additional power reduction as we will show in Chapter 5.

1.2 Android Application Characteristics

In this section, we describe Android-specific characteristics of applications. More concrete, we focus on applications states that can be found across not only one but several different kinds of applications. Then, we describe common performance metrics that do not only apply for Android apps but for all interactive applications.

1.2.1 Application Performance States

The types of interactions between a user and his or her smartphone are manifold. While there are actions performed by the user that require immediate response by the device, other tasks or actions are less critical. For example, applications are expected to respond immediately to a touch event like scrolling or zooming, and maintain a certain frame rate during animations. However, there also exist background tasks that have little impact on the user experience. Maximum reaction times have already been part of the research performed by the Human-Computer Interaction (HCI) community [108] and have been adopted for specific applications, such as in the RAIL model for the web browser [47] (described in more detail in Section 5.4.1). In the following, we take a cue from these models and describe how applications can be divided into states, based on their performance requirements.

A key characteristic of applications running on mobile devices is that they are user experience-sensitive. Figure 1.5 shows typical possible transitions between these *states* in Android applications. After startup, many applications wait for input from the user. As the input can be delayed for various reasons, there are often states where the device is waiting for an input and is doing nothing in the meantime. This is a typical *idle* state. However, we can differentiate between *busy idle* and *true idle* states. During true idling, the system appears idle to the user and there

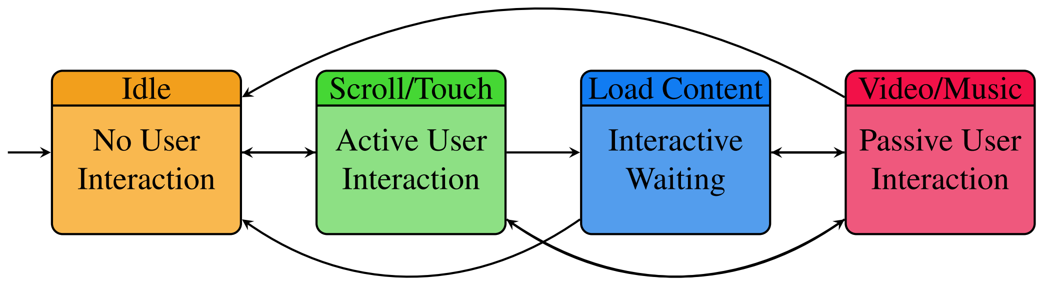


Figure 1.5: Example for different Android application states.

is no workload being processed by the system. However, during busy idling the system might appear idle, but there are background tasks being executed that are not perceivable by the user.

While the previously described states do not involve any interaction with the user, there are multiple types of *interactive* states that differ among their performance requirements. All interactive states have in common that the user's action usually requires timely response from the system to guarantee a good user experience. The types of interaction vary between simply touching the screen to choose an object, and scrolling through the screen or zooming. After an interaction with the device, the user is often waiting for content to be loaded, such as for an application to start up or for a web page to appear when browsing the web. Further, another way of interacting with the device is watching a video or listening to music. Compared to scrolling or typing, which are *actively* done by the user, watching a video can be considered a *passive* interaction. While there has been a lot of work on video power management previously, application-aware power management is shifting towards the active type of interactive applications, especially on mobile devices.

As mentioned above, all the different states also have different performance requirements. Idle states usually do not require high computation power because there is no need to react fast to possible user actions. However, if a user action has occurred, the highest priority of the system is to respond to that particular action in an acceptable amount of time. An optimal power management strategy consumes the least amount of power for the best possible performance.

1.2.2 Performance Requirements among Application States

By considering the different application states and the degrees of interaction between the user and the applications, it comes naturally that the CPU workload distribution among the different states varies. While the workload for idle states is expected to be low, we expect the workload for user interactive phases to be considerably higher. There are mainly two different metrics for interactive states that define the user perception of a particular action: The *frame rate* and the *response time*. As mentioned above, such metrics are derived from the HCI community [108] and adopted to a specific application, such as the RAIL model [47] for the web browser.

The frame rate is usually measured in FPS. A typical target value for the frame rate is 60 FPS, a value that is derived from the Vertical Synchronization (VSync) signal of the display. The VSync signal is issued by the display to trigger the redraw of the screen at a rate of 60 Hz.

However, research has shown that it is also possible to achieve good user perception results at as low as 30 FPS [27]. The response time should be kept as low as possible to guarantee the best possible user experience. The term does not only refer to the time that an application should react to a given user input but also, e.g., to the loading time of a web page. For browsing, the maximum response time to a user action for an already loaded web page is 100 ms, while the maximum loading and rendering time of web pages is 1 s for desktop computers and up to 5 s for mobile devices, where slower data transfer rates are tolerated. During idle states, the goal is to keep the power consumption of the system as low as possible. However, the system has to maintain responsiveness to user events that can occur suddenly and invoke a transition to an interactive state.

The two metrics frame rate and response time vary significantly in terms of resource requirements to satisfy the performance requirements. For the frame rate, it is only necessary to provide the amount of power needed to meet the target FPS value. The response time, however, should be as low as possible, what can only be guaranteed by providing the maximum resources available.

1.2.3 State-Based Android Power Management

Given the states defined above and the corresponding performance requirements, it comes naturally that the states can be exploited for power management. The idea of state-based power management for games has been introduced before [35] and has been proven as effective. In [35], a game state and a loading state for a number of games have been defined. The frame rate of the game is adjusted according to the state to save power. Moreover, the frame rate has been taken into account to perform DVFS and to choose a CPU frequency that would just fulfill the desired frame rate. Similar to games, states can be found in other applications, as we will show in Chapter 5 and Chapter 6.

1.3 Application-Aware Android Power Management

While we have previously described different performance requirements for Android applications, we explain the different layers, e.g., kernel and applications, at which power management can be performed in Android. Then, we describe common challenges that have to be overcome when performing power management across different layers.

1.3.1 Levels of Power Management in Android

As mentioned in Section 1.1.4, the Android power management has a lot of potential for improvement, especially on HMP platforms. Although the Android default governors do an overall fair job across a wide range of applications, there are a lot of scenarios where an optimization towards power consumption is possible.

To perform power-performance optimization, some knowledge about the underlying hardware platform and also about the characteristics of the application is needed, as depicted in Figure 1.6. The main difficulty is to analyze how many resources an application really needs

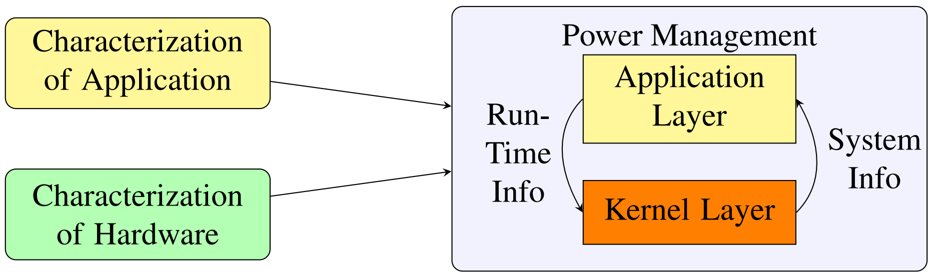


Figure 1.6: General approach to power management.

to execute without any distortions. This not only depends on the application but also varies between different hardware platforms. While a game might need only 3 ms to compute a frame on a powerful CPU, it might take twice as much time on a less powerful platform. Finding the optimal configuration for a given platform is one of the main challenges for application-aware power management. Therefore, it is important to analyze the application and to look for hints that allow us to perform power management across different platforms.

Single-Layer Power Management

As shown in Figure 1.6, power management can involve different software *layers*. First, it is possible to perform power management only at the kernel layer, as currently done by the Android default governors. Here, it is sufficient to maintain a minimum of information, in case of the default governors the CPU utilization, and also some basic knowledge about the hardware platform to perform DVFS. Second, it is possible to perform power management solely at the application layer. Such techniques include, for example, targeting a display frame rate of 30 FPS instead of 60 FPS for animations or the usage of a backend server for heavy computations instead of performing those computations locally. All methods mentioned above do not involve any interaction between the different system layers, namely the applications and the kernel.

Cross-Layer Power Management

Involving communication between the layers increases the complexity of the system, but allows for more effective power management techniques. The direction of communication can be either from the application to the kernel or vice versa. An example is an application that shares its internal information to the kernel, e.g., whether a game that is currently loading, or the frame rate of a running animation (see [35, 126]). The kernel can also give hints about its internal state to the application, e.g., to influence the number of threads that are currently spawned by an application by providing the number and usage of the CPU cores. Generally, it is also possible for the application and the kernel to provide information to each other simultaneously.

Hybrid Power Management Techniques

Another option is a mixture of the two former strategies: The single layers perform power management on their own, but can also share information with the other layer. For such constellations, it is important that the power management techniques applied by the different layers are either orthogonal to each other or coordinated among each other. For example, a technique where the the application reduces the frame rate is independent from the kernel doing power management on a workload-based manner. Here, the system can profit from two independent strategies. However, if an application produces a number of working threads dependent on information from the system but the system turns down the corresponding cores, this would most likely result in a degradation of the user experience. Hence, the applied strategies should not interfere with other possible approaches.

1.3.2 Challenges of Application-Aware Power Management

When performing application-aware cross-layer power management, there are many multi-faceted challenges that have to be considered. For example, there is a trade-off between the efficiency of an application-aware technique and its applicability to other applications. Another important point is the awareness of the underlying hardware platform. Further, the overhead of the method should be as low as possible, both in terms of time and in terms of power.

Power Management Overhead

While the Android default power managers only consider the workload, application-aware power management adds significant overhead to the power manager itself: The communication between the kernel and the application layer. The communication should be implemented such that it does not increase the execution time of the application (e.g., by blocking write or read calls). Furthermore, the power management overhead itself depends on the actual complexity of the power management technique, e.g., the complexity of the algorithm. While a moving average workload predictor adds only a light overhead, an integral-based predictor will most likely need significantly more computation time. We do not investigate the overhead of our implementations, because our implemented strategies perform significantly better in terms of power consumption than the default strategies.

Generalization across Applications

While application-aware techniques that consider unique information from one particular application are usually more effective in terms of power savings, the downside is that those techniques can be used for one specific application only. On the other hand, more general techniques tend not to cover distinctive features of specific applications. This can result in a better performance for the whole system but in a worse performance for one specific application.

However, the Android power managers do not consider any application-specific information at all. As shown above, there are repetitive states across different Android applications that have similar performance requirements. Hence, we believe that there is potential to do application-aware power management across different applications much more efficiently than is it currently implemented. In order to target this problem, we will study different applications and derive a common API from the results (see Chapter 6). We start by studying game power management on HMP platforms (see Chapter 3), as Dietrich et al. have already provided a fundamental baseline on this topic [33, 35, 36]. Then, we look into browser power management because browsing provides a variety of states and actions that can also be found in other applications (see Chapters 4 and 5).

Platform-Aware Android Power Management

As mentioned before, another important factor for effective power management is the platform-awareness of the power manager. This holds true in particular for power managers that operate at the system layer. We have explained in Section 1.1.3 that Android has three power management entities - the governor, the scheduler and the power control - that work independently from each other and how this is problematic on modern HMP platforms.

We believe that the interplay between the separate components of the power management units would improve the power consumption of the system significantly. For example, if the power control unit knew whether tasks should be currently scheduled on particular cores or CPUs, it could keep unused cores or CPUs in a low-power state. On the other hand, the power control unit could turn on cores or CPUs if it realized that more computation power is required. Moreover, the scheduler and the governor would profit from sharing information. While the governor maintains information about the available frequencies of the system, the scheduler knows all the active threads that will be running in the next scheduling instant. The scheduler also can provide information about the workload of the available threads. All of these information could be combined to find the optimal mapping of threads to cores such that the CPU frequency can be minimized. We have investigated the Odroid-XU3 platform very thoroughly to find the best configurations for the optimal power management technique, as will be described in the Chapters 3, 4 and 5.

Cross-Platform Android Power Management

When the application and the kernel interact, there must be a defined method specifying how the power managers best react to changes in the application, to avoid undefined behavior by the power managers. This poses a challenge that has been already addressed by the default Android power managers: Cross-platform power management. While the Android default governors only need a little fine-tuning to work with different hardware platforms, an involved application-specific governor might need much more fine-tuned algorithms to achieve the best possible power savings. Hence, it is a major challenge to design a governor which is easily portable across multiple platforms and also provides a general interface for a multitude of applications. We will detail the challenges of the platform dependency in Chapter 6.

1.4 Contributions

The main focus of this work lies on mobile game and browser power management for Android. Based on these parts of our work, we derive a generalized power manager that can be applied not only to specific applications, such as games or browsers, but that can be applied to any Android application. The main contributions of this work are summarized based on the application they focus on:

Game Power Management:

- We characterize the gaming thread workloads and develop a thread-based and frame-based hybrid workload predictor to accurately predict the gaming workload.

- We implement an integrated power manager, which we refer to as *GameOptimized governor* in the following, and compare it to the default Android governors.

- We perform a user study to evaluate the user perception of our proposed power manager.

Browser Power Management:

- In the area of mobile web browsing, we give a detailed analysis and characterization of the mobile web browser workload for loading a web page by breaking down the browser CPU time and CPU energy based on the main browser processes and their threads for a number of representative web pages.

- Based on the non-trivial analysis, we look into potentials of power saving for mobile web browsing workloads on HMP platforms using core allocation of individual threads, DVFS, and power gating of the A15 CPU.

- We outline potential power saving techniques, such as the need for an integrated power management unit on HMP platforms which combines scheduler, governor, and power control.

- Further, 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 and demonstrate the effectiveness of this approach in terms of power consumption as well as responsiveness of the system.

General Android Power Management:

- Finally, we introduce a generalized power management API as a communication channel between the kernel and the applications, that we derive from our previous work on game and browser power management.

1.5 Organization

This thesis is organized in seven chapters. In this first chapter, we introduced the reader to the topic and gave an overview of Android applications' characteristics and Android CPU power management techniques on a modern smartphone architecture which motivated this work. The remaining document is structured as follows:

Chapter 2 outlines other important contributions related to application-aware Android power management, where we mostly focus on CPU power management. In particular, we give an overview of video power management, which gave important cues for mobile game power management. Moreover, we introduce existing work in the area of mobile browser power management - some of which we took as motivation for this particular work. Finally, we outline more general Android power management strategies related to other peripheral components, that consume a significant amount of power besides the CPU and deserve some attention when addressing the topic of Android power management, namely the display and the wireless link.

Chapter 3 shows the importance of workload prediction for mobile games and that this holds true for HMP platforms in particular. In this chapter, we emphasize the importance of sharing information (here: frame rate) between the application and the power manager to perform better power management. We extend the previous frame-based power management techniques to HMP platforms to propose a frame-based and thread-based, predictive power manager for mobile games. The proposed framework is capable of performing thread allocation and DVFS simultaneously to meet the FPS requirement of the user while minimizing the power consumption. We characterize the gaming thread workloads and implement a thread-based and frame-based hybrid gaming workload predictor. The hybrid predictor learns online whether the thread workload is periodic or aperiodic and the power manager, which we refer to as *GameOptimized governor*, performs thread-to-core allocation and DVFS simultaneously, based on the predictions. We have implemented the GameOptimized governor on our evaluation board and compare it to the default Android governors. Moreover, we performed a user study to evaluate the user perception of our proposed power manager and find that up to 60.0% of energy can be saved for using power manager, while the user perception is considered good. The results of this work have appeared in [123].

In Chapter 4, we investigated the power management potentials of the web browser. While there have been many works on game workload characterization and game power management before, web browsing has been less studied, especially with respect to Android application states and power management potentials on HMP platforms. To evaluate these potentials, this chapter provides a non-trivial, detailed analysis of the actual thread workloads generated by the web browser for a number of web pages on the Odroid-XU3 platform. We give a detailed analysis and characterization of the mobile web browser workload for loading a web page by breaking down the browser's CPU time and CPU energy based on the main browser processes and their threads for representative web pages.

We identify the process consuming most of the energy, which is the renderer, and further break down its energy consumption for different website components. Based on the analysis, we look into the potential of power savings for mobile web browsing workloads on HMP platforms using core allocation of individual threads, DVFS, and power gating. Further, we show in a first attempt that we can save up to 39.21% of the CPU power consumption when we power gate the big CPU after a web page has finished loading. We also introduce our measurement infrastructure that we have developed for logging all the performance and power relevant information such as the core utilization, CPU frequency, power consumption, thread allocation, function tracing, etc., for the underlying HMP hard- and software platform. This infrastructure is a prerequisite for all analysis and characterization work as it enables us to find which CPU a thread is scheduled on - the most relevant information in the case of HMP systems. The results of this work have appeared in [124].

In Chapter 5, we have exploited the results and clues on the browser power consumption obtained in [124] to develop a new phase-aware power manager for web browsers on HMP platforms. We show how sharing information between the power manager and the running application is beneficial for power management. First, we define different browsing *phases*, such as *Idle*, *Load*, *Scroll*, etc., which exhibit distinct workload characteristics and user requirements. These phases are based on the application states that can be found in Android. Then, 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 cpufreq Linux kernel governor – the browser governor – that controls the CPU power state and its voltage and frequency according to the available phase information based on a power management strategy that we have defined per phase. Once again, we evaluate our approach on the Odroid-XU3 platform and demonstrate its effectiveness in terms of power consumption as well as responsiveness of the system. The results of this work have appeared in [126].

Based on the results that we have found in our previous works [35, 123, 124, 126], we propose an API between the power manager residing in the kernel and Android applications within the user space in Chapter 6. From the example scenarios gaming and browsing, we can derive many use cases that can also be found in other types of Android applications. We propose an API design where information such as the current workload priority (e.g., high or low), the frame rate (current and target frame rate) and a deadline (e.g., the workload x has to be completed until time y) can be provided to the kernel. We also explain the challenges for kernel and application developers when introducing such an API. This work has partly been published in [125].

We summarize our results and conclude this work in Chapter 7. In this chapter, we outline the potentials of further research in the area of Android power management, especially browser power management and Android power management in general. In particular, we suggest open research questions for the given topics.

1.6 List of Publications

Parts of the work reported in this thesis appear in the following papers. Those publications form the basis of this thesis. In particular, the thesis is based on:

- **Nadja Peters**, Sangyoung Park, Daniel Clifford, Sami Kyostila, Ross McIlroy, Benedikt Meurer, Hannes Payer, Samarjit Chakraborty, *Phase-Aware Web Browser Power Management on HMP Platforms*, in International Conference on Supercomputing (ICS), Beijing, China, 2018.

- **Nadja Peters**, Sangyoung Park, Daniel Clifford, Sami Kyostila, Ross McIlroy, Benedikt Meurer, Hannes Payer, Samarjit Chakraborty, *API for Power-Aware Application Design on Mobile Systems*, in International Conference on Mobile Software Engineering and Systems (MobileSoft), Gothenborg, Sweden, 2018.

- **Nadja Peters**, Sangyoung Park, Samarjit Chakraborty, Hannes Payer, Benedikt Meurer, Daniel Clifford, *Web Browser Workload Characterization for Power Management on HMP Platforms*, in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, USA, 2016.

- **Nadja Peters**, Sangyoung Park, Dominik Füß, Samarjit Chakraborty, *Frame-based and Thread-based Power Management for Mobile Games on HMP Platforms*, in International Conference on Computer Design (ICCD), Phoenix, USA, 2016.

The papers listed below are related to the general area of power management:

- Benedikt Dietrich, **Nadja Peters**, Sangyoung Park, Samarjit Chakraborty, *Estimating the Limits of CPU Power Management for Mobile Games*, in International Conference on Computer Design (ICCD), Boston, USA, 2017.

- Philipp Kindt, Jing Han, **Nadja Peters**, Samarjit Chakraborty, *ExPerio - Exploiting Periodicity for Opportunistic Energy-Efficient Data Transmission*, in Conference on Computer Communications (INFOCOM), Hong Kong, Hong Kong, 2015.

2 Related Work

Due to the increasing importance of battery runtime of mobile devices, Android power management has received a lot of attention not only in research but also in the industry during the last years. In this chapter, we provide an overview of the most relevant topics in regard to Android power management. The power distribution in a smartphone highly depends on the underlying hardware platform and differs among the variety of available smartphones on the market. Hence, it is challenging to provide numbers for the power consumption of mobile devices. First, the power consumption highly depends on the hardware platform and second, the power consumption of individual components also highly depends on the usage scenario. However, there are some components that usually consume more power than others. For example, the display is one of the major contributors towards smartphone power consumption, especially when brightly illuminated [14]. Moreover, the wireless link, in particular the Global System for Mobile Communications (GSM) module, has a high power dissipation when used heavily. Nowadays, the CPUs have become capable of performing complex calculations in a short time frame. Hence, their power consumption is steadily increasing although industry is working hard on finding a balanced solution between performance and power consumption, for example the big.LITTLE platform, designed by ARM [92].

Although our work mainly focuses on CPU power management, we consider it very important to give a broad overview of related topics such as display and wireless link power management. This enables us to put our work into a larger context by taking cues from other research areas. First, we give an overview of CPU power management, in particular over application-aware power management in Section 2.1. We focus on video, game and web browser power management, as the most relevant applications related to our work. In Section 2.2, we introduce power management related to the wireless link components of mobile devices. As there exist a number of different protocols for data transmission, we have divided the section into 3G, 4G and WiFi power management as well as hybrid protocols. Finally, we cover display power management in Section 2.3. Here, we make a distinction between two state-of-the-art technologies: Liquid Crystal Display (LCD) and Organic Light-Emitting Diode (OLED) displays.

2.1 Application-Aware CPU Power Management

While the default Android governors are ignorant of the currently running application, there are many works that have developed approaches to perform power management for specific applications. We will introduce some of these works in the following. Mainly, we focus on video power management, game power management and web browser power management. These applications are the most popular in terms of application-specific power management. However, we also introduce some more general approaches for application-aware power management in the following. As graphics contribute a lot towards game power consumption, we will also cover some Graphics Processing Unit (GPU)-related approaches in Section 2.1.2.

2.1.1 Video Power Management

Video power management dates back a long time, before the era of smartphones had begun. Although videos are passive interactive applications, video power management has inspired the work of game power management significantly. Moreover, videos are one of the most important and popular media formats, nowadays. Only as an example, the popular YouTube platform has over one billion users and is even more popular than any television network for 18 to 34 years old inhabitants of the USA [163]. As there has been a lot of work on video power management over the years, we will focus on CPU-related DVFS strategies in the following.

As mentioned above, video power management has been the focus of research for a long period of time. Video streams consist of a finite number of video frames that have to be processed such that the video can be shown on the screen. Typical steps during this process are video decoding, transformations or motion compensation. Depending on the actual frame, the workload to perform those steps can differ - which is the key prerequisite for performing DVFS. For constant workloads, the frequency can simply be fixed to a particular value. For completeness reasons, it should be noted that DVFS was often referred to as Dynamic Voltage Scaling (DVS) in the past, as can be read in some of the related work papers. However, the functionality of DVS and DVFS is similar for the works introduced in this section. Hence, we will only use the term DVFS for the reason of consistency.

There have been a number of earlier works that explore video power management [122, 69, 70, 24, 165, 130, 96]. Pering et al. looked into different types of applications on a Personal Digital Assistent (PDA), of which one is video decompression, in particular Moving Picture Experts Group (MPEG) decoding [122]. The findings are 1) that the video workload is of a varying nature and therefore it is suitable for DVFS. Moreover, 2) their test results show that by applying a moving average algorithm 24 % of power can be saved. They also performed a simulation for the theoretical optimal power savings and found that they could even save up to 60 % by applying a more advanced technique. Hudges et al. presented different studies that looked into DVFS and how to apply it to video power management [69, 70]. Besides audio, these works focus on H.263 decoding. The authors found that the workload of the decoder is highly variable and, hence, suitable for DVFS. They defined the workload as a function of the Instruction Count (IC) and the Instructions per Cycle (IPC). The IC depends on the application's algorithm and the input to the application, while the IPC on the algorithm, input, and the architecture of the system. By looking at consecutive frames, the authors found a workload correlation between neighboring frames for videos, which could be exploited for power management. This correlation can be used for the prediction of the workload, which itself can be used to pick the appropriate CPU frequency to process a video frame. Choi et al. presented another work, which looks into frame-based prediction of MPEG workload [24]. The authors divided the workload into a frame dependent and a frame independent part. While the workload for the frame dependent part is predicted using a moving average predictor, the workload of the frame independent part remains constant. The approach was implemented and evaluated on the StrongARM evaluation board. The authors reported a 16 % energy reduction for the total system. A combined scheme of task scheduling and DVFS was presented by Yuan et al., that is called *GRACE-OS* [165]. The Linux kernel was modified by adding a profiler that monitors the workload of multimedia tasks, calculates or predicts the future workload. Based on the predictor, a Soft Real-Time (SRT) scheduler decides the timing of the tasks and a speed adapter adjust the CPU frequency. The prediction is calculated online, based on the actual system workload. This approach was further improved and extended resulting in the *EScheduler* [166]. Akyol et al. applied a complexity model that takes into account the video source, the video encoding algorithm and the hardware platform specifications [3]. They estimated the future workload of video decoding tasks based on Normalized Linear Mean Square (NLMS) predictors. While the previously introduced approaches mainly focused on systems with CPUs where the single CPU cores share the same frequency, Khan et el. evaluated a system with individually tunable CPU cores [77]. They achieved an average of 39 % power savings using their approach.

While the previously presented works rely on workload information per frame obtained from the system, Pouwelse et al. introduced a technique that suggests providing the relevant workload information directly by the application [130]. A video decoder was modified in such a way that it could predict the future workload based on the frame size and the frame type of the particular frame. Based on the prediction, the application itself could perform a system call and change the CPU frequency setting as needed. The approach is more effective in terms of power consumption compared to a statically fixed frequency approach and also to a dynamic interval-based scheduler that responds to the average workload. Another work by Lorch et al. implemented PACE (Processor Acceleration to Conserve Energy) that defines deadlines for particular tasks and re-schedules tasks to improve the power consumption of the system [96]. Moreover, they proposed a race-to-halt strategy for tasks that miss their deadline. However, recent work has shown that race-to-halt is a very energy consuming strategy for state-of-the-art SoCs [123, 124]. Huang et al. also suggested providing hints about the computation demand of decoding a video to the power manager [67]. However, the approach is based on an offline watermarking technique. They analyzed the video stream before transferring the video onto a PDA device and marked the frequency that is necessary to compute the frame in time. The information was inserted into the video stream and could be read on the particular device for frequency scaling. Their workload prediction is based on so-called macroblocks, that MPEG-2 video is built from. Each block itself consists of three tasks that significantly contribute to the workload:

The Variable Length Decoding (VLD) task, the Inverse Discrete Cosine Transformation (IDCT) task and the Motion Compensation (MC) task. The main advantage of this approach is that the workload prediction is not performed on the mobile device. This saves a considerable amount of energy and, hence, battery run time. Sultan et al. proposed a new approach to save power by introducing different video stream decoding strategies based on the current battery state of the device [151]. They suggested to decrease the quality of the video step by step. For example, the best video quality is given at full Signal-to-Noise Ratio (SNR) values, with full spatial resolution and at full frame rate. As the battery level falls, all of these values can be decreased stepwise. First, the frame rate can be decreased from usually 30 to 15 FPS. Later, the spatial resolution and the number of SNR values can also be dropped. Although this strategy is highly effective in reducing the power consumption of up to 86 %, it has been shown that dropping the FPS to such a low rate does affect the user perception of the animation [26, 27]. Hence, it is preferable to apply methods that reduce the power without impacting the quality of the video.

While the previous approaches mainly focus on video decoding, there have been attempts to reduce the power consumption of video encoding as well. Jin et al. presented a DVFS scheme that applies Hilbert Transform-based Workload Estimation (HTWE) for workload prediction [76]. They evaluated their approach using a PC platform and a Hitachi Evaluation board and report power savings of up to 61.69 % for their approach. Another approach for MPSoCs was presented by Iranfar et el. [72]. They suggested applying machine learning for power and thermal management and use data from frame compression, quality, performance, total power and temperature as learning parameters.

2.1.2 Game Power Management

Nowadays, mobile games are a popular kind of application as smartphone users tend to spend a lot of time gaming. However, games belong to the most power hungry applications on mobile devices and power management is challenging due to the interactive nature and the complexity of modern games. We will outline the major works in the field of game power management in this section. However, we will explain the main advantages of our work over the related works in the further chapters, in particular Chapter 3.

Lin et al. were one of the first that proposed DVFS for power management in interactive applications [93, 100]. In this study, the authors introduced a user-centric feedback mechanism to control the frequency of the processor. While an application is running, the user is able to provide feedback about his or her perception. As testbed, a Windows XP Laptop was used. The CPU features four different frequency levels and the default Windows DVFS policy is purely workload-based. During a user study, the participants played (among other applications) a FIFA game to test the implementation. A comparison between the Windows default strategy and the implemented DVFS policy resulted in 22.1 % power savings. However, the interaction between the user and the power manager adds an additional burden to the user and makes the game less enjoyable. Hence, a built-in power manager that does not require the input of the user is preferable.

Gu et al. started the first line of work on DVFS for game power management that was independent of the user and is based on workload prediction [50, 51, 52, 53]. Starting in [53], the authors analyzed the open-source game Quake II running on an IBM notebook to identify patterns among game frames and whether those are applicable for workload prediction. They identified that the different components of the game, e.g., the number of polygons, the rasterizing of textures or particles, underly particular models or can be approximated by the constituting pixels. Several DVFS schemes based on workload prediction are presented in [50, 51, 52]. A workload predictor that is based on the model in [53] was introduced in [52]. For each so-called *view frustum* - the part of the modeled environment that will be visible to the user on the screen - the workload can be calculated by applying the model. The main drawback of this method is that the workload calculation involves all objects that should be rendered to the screen and creates a large calculation overhead. Despite the overhead, the authors reported up to 50 % power savings on their testbed. In [50], the group presents a Proportional Integral Derivative (PID) controller for the previously mentioned Quake II game that is used to predict the workload of the future game frames. The controller achieves considerable power savings compared to the default frequency governor - up to 22 % - but the PID parameters have to be tuned manually. Hence, the approach lacks portability to other (non open-source) games. Finally, the group implemented a hybrid controller that combines the two previous approaches and tested their approach on a PDA, using Quake as their test application. The results show that the hybrid approach yields the best results on the PDA.

Similar to videos, games are computed on a per frame basis. While recent games often target 60 FPS, it has been shown by Claypool et al. that a good user experience can be obtained even with 30 FPS [26, 27]. The authors did not only look into different frame rates but also into different screen resolutions and their effects on the ego shooter game Quake. The group looked at how different frame rates and resolutions affect the perceived quality of the game play by the user. A user study showed, that, surprisingly, the screen resolution had almost no effect on the game play. However, the FPS plays an important role and a value below 30 FPS has a negative impact on the user performance. These studies [26, 27] have a large impact, as many newer power management strategies target 30 rather than 60 FPS to minimize the power consumption [35, 111].

Lowering the frame rate is not only beneficial in terms of CPU but also in terms of GPU power consumption, as was pointed out in [111, 162]. Since the game frame workload and therefore the required processing frequency highly varies from frame to frame, research mainly focused on identifying techniques that allow predicting the future workload of game frames.

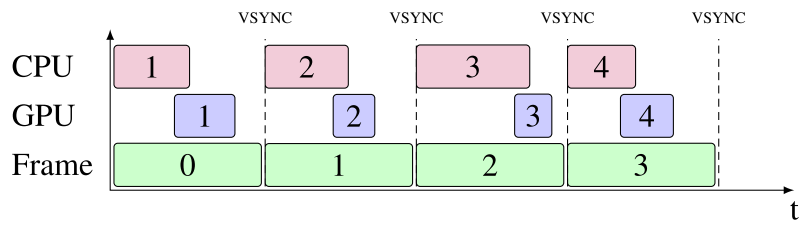


Figure 2.1: Typical frame timing [35].

Figure 2.1 shows how frames are calculated during the game. Nowadays, each frame consists of some calculations done by the CPU. Then the actual graphics processing and rendering the frame to the screen is done by the GPU. In case of games, the CPU calculation is usually based on the Artificial Intelligence (AI), the current positions of the objects, the game logic and potential user inputs. Different frames can exhibit different complexities, depending on the previously mentioned aspects. The calculation time is usually set by the VSync signal of the display that is issued every 60 Hz, but software-wise it is also possible to skip signals and reduce the frame rate, e.g., to 30 Hz. Previous research has shown that sequential frames (in terms of time) usually exhibit a similar workload [37]. This can be exploited to develop workload predictors and adjust the CPU frequency according to the predicted CPU workload. However, in order to apply DVFS, it is crucial to estimate the frame workload accurately.

This motivated Dietrich et al. to develop sophisticated workload predictors [35, 36, 37], which enabled DVFS-based power management and outperformed the default Android governors in terms of power consumption and exhibits decent performance. For example, in [36], it has been shown that the workload of neighboring frames correlates, which can be exploited for fine-grained power management. However, the frame workload prediction accuracy is never 100%, and often under- or over-estimated. This leads to frame drops and sub-optimal power savings. Yet, it is difficult to analyze how close to optimal the state-of-the-art techniques are, and therefore, it is hard to judge whether more sophisticated techniques such as non-linear workload predictors are worth investigating. An effort to investigate the potential of an optimal predictor was made in [32]. In order to address the problem, the group designed a statistical model based on power and workload measurement conducted on an experimental Android platform. They found that the theoretically possible optimum for DVFS is up to 54 % more power savings compared to previously published game power managers for the given platform. Given these numbers, future endeavors in the field of game power management seem promising.

With the emergence of multi-core CPUs and even heterogeneous multi-processing platforms that combine low-power and high-performance CPUs on a single chip, the multi-threading potential of games has been investigated, recently [118]. Pathania et al. presented a line of work that deals with game power management on multi-CPU platforms and also takes the GPU into account. [118] contains a detailed investigation about the actual workload distribution among the threads in games. The work describes the scheduling of different threads and the frequency scaling of the CPUs as a problem that is based on a CPU workload-capacity model. Each thread costs a price in terms of workload. Starting at a configuration where the little CPU is at its lowest frequency level, the threads are mapped to CPU cores as long as they provide sufficient workload capacity. If not, the frequency of the CPU is increased or the thread is transferred to the big CPU. This guarantees that the CPUs are operated at the lowest frequency level with maximum utilization, which is more power-efficient than operating the CPUs at high frequencies.

Besides CPUs, GPUs have become an integral part of modern SoCs, recently [99, 116, 117, 2, 61]. As mentioned above, Pathania et al. conducted experiments in which they compare the CPU and the GPU workloads, dependent on the frequency of CPU and GPU, respectively [116, 117] . They showed that some games are more CPU bound while others have a performance bottleneck at the GPU side. More precisely, this means that the number of rendered game frames will increase significantly with the increase of the GPU frequency for some games while for others the FPS will increase linear with the CPU frequency. There are also games with hybrid CPU-GPU dependencies. Again, the authors created a cost model, that also involves the GPU. Based on the required FPS of the game, the model picks an appropriate CPU-GPU configuration. This approach works reactive rather than predictive, but still achieves respectable power savings. A more recent work by Ho et al. also deals with interactive applications and takes into account CPU and GPU DVFS [61].

2.1.3 Web Browser Power Management

In this section, we introduce recent works that deal with web browser power management. While there has been numerous works targeting the performance of the browser [160, 144, 12, 28, 31, 54, 105, 81], that should at least be mentioned for the sake of completeness, we will focus on studies that are related to power management.

Thiagarajan et al. presented one of the most inspiring works for our research. They analyzed the power consumption of different components of a web page, such as JavaScript, Cascading Style Sheets (CSS) and images [154]. The work shows that up to 50% of the rendering energy of a web page is due to JavaScipt - depending on the web page. We took a cue from [154] to investigate the power consumption of the web browser in our own work [124], presented in Chapter 4. Besides the characterization, the group also introduced some methods to save energy while browsing the web. For example, they suggested to modify web pages such that only Joint Photographic Experts Group (JPEG) images are used rather than Portable Network Graphics (PNG) images, because those can be compressed and rendered with less overhead. Moreover, they demonstrated the effectiveness of computation offloading to remote servers.

A very detailed analysis of the web browser power consumption was presented in [13]. Cao et al. introduced a detailed model of mobile web page loading, namely *RECON*. RECON can estimate the energy consumption of a web page load based on a linear regression model. The approach was validated on multiple hardware platforms and found to have a mean error of 6.4 % for an entire web page load. The main application of RECON is to find energy pitfalls during web page development. For example, the authors found that some adblockers consume more energy for web pages without ads, although the loading time of those pages is not increased. Zhu et al. presented an analysis of the relationship between the transmission data rate and the CPU power consumption [170]. They found that the CPU workload increases with higher data rates while there are a lot of idle times for low data rates. Hence, it is possible to either adapt the CPU frequency to the corresponding data rate. On the other hand, they also suggested that it is possible to adapt the network speed to the computation speed of the CPU to avoid power wastage on the wireless link side.

A series of work on power management for mobile web browsing was introduced by the same group around Yahao Zhu [169, 171, 172]. They presented a study about the impact of CSS and Hypertext Markup Language (HTML) tags on the web page loading time [171, 172]. In [172], they developed a model that is based on these tags and utilized it to regulate the CPU frequency of their experimental board. This work was continued in [171], where Zhu et al. developed a model that is also based on HTML and CSS tags but is used to determine whether the browser should be scheduled on the big or the little cores. The big.LITTLE platform in [171] is not a single platform containing an integrated SoC with two CPUs. It rather consists of two separate hardware platforms being evaluated separately. In parallel, the group released a work where they studied the effect of CSS, HTML and JavaScript on the performance and energy consumption of web page loading on a desktop computer [172]. The latest publication in this line of work introduced *eQos* [169]. The approach profiles user and system events to identify the QoS required by a mobile web application. The data is used to perform CPU task allocation and DVFS on a big.LITTLE platform. Different applications are used to verify the approach, for example zlib and the Google Chrome browser. The authors used indirect information of the context, e.g., web page primitives or user events - for their strategy. They did not use information that are provided directly by the application itself. For eQos, the underlying hardware platform is a real big.LITTLE platform, namely the Odroid-XU+E board [59]. Compared to the platform that we have used in our works, this is not an HMP platform, because only one of the CPU clusters, either the little or the big one, can be used at the same time.

Another area of web browser power management is the power consumption that arises due to advertisements or *ads*. Recent works have shown that although advertisements mostly contribute to a higher data usage and CPU power consumption, there also exist adblockers that actually raise the power consumption of the web browser [13]. As depicted in Figure 2.2, there is a basic power consumption for loading a web page, which increases when advertisements are displayed. However, there is also an overhead using the adblocker. If ads on a web page do not consume additional energy compared to loading the baseline web page, the overhead of the adblocker might actually be higher than running the web browser without an adblocker. Cao et al. showed that web pages without intrusive behavior, such as playing a video, often consume less power without an adblocker than applying the adblocker called *BSDgeek_Jake* [13].

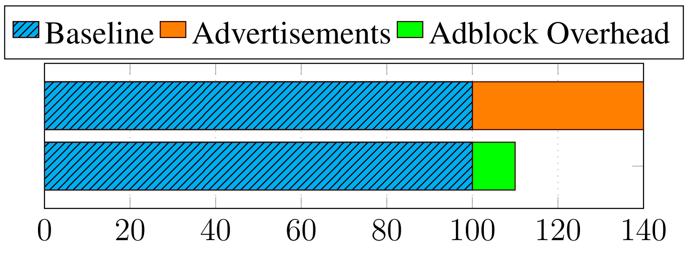


Figure 2.2: Energy consumption with and without adblocking [129].

There has been a number of interesting works that cover the energy consumption of advertisements in mobile apps in general [155, 56, 55]. However, given the importance of the topic, we have focused on the energy consumption of web browsers caused by the contents of the web page. Simons et al. studied the power consumption of adblockers and tracking protection across different browsers on different Windows desktop computers [148]. They found that advertisements contribute 3.4 % towards the total energy consumption of the computers. Rasmussen et al. tested the total phone energy consumption of an Android phone based on different adblockers [135]. They found that the energy consumption was highly dependent on different host files that were used by the adblocker. The best improvement that they found is 3.8 % and they report that the energy consumption can even increase up to 6.5 % for some of the host files. A more recent work by Visser et al. used an external adblocker rather than a build-in adblocker in the browser to quantify the energy overhead of advertisements [156]. The authors found that advertisements consume roughly 27 % of the average energy consumption on an Samsung Galaxy S5 smartphone.

2.2 Power Management for Wireless Link

An important question in Android power management and especially for web browsing but also all other application that rely on a functioning Internet connection is the power consumption and the performance of the wireless link. There has been a large number of studies that investigated the power consumption of the wireless link. Most works focus on the power consumption of either the 3G or Universal Mobile Telecommunications System (UTMS) interface [8, 65, 119, 104, 62, 83, 168], the 4G or Long Term Evolution (LTE) interface [63, 64, 107, 57, 90, 86, 161], the WiFi protocol [1, 167, 101, 120, 17, 89, 9, 136, 170] or a combination of the former [121, 133, 110, 49, 109, 58, 173]. We present the most relevant works in the following.

2.2.1 3G Protocol

As one of the first, Huang et al. provided significant insight about how the 3G network and its performance affects the user experience on mobile devices by introducing their tool *3GTest* [65]. The tool looks into web browsing, video streaming and Voice over IP (VoIP) applications, and identifies network problems, performance bottlenecks on the phone or related to the contents. The authors collected data from around 30000 users around world and presented a variety of results for different carriers and phone types. Hu et al. analyzed the 3G traffic and performance in China and complemented the previous work [62]. Other works looked more closely at the 3G transmission protocol [8, 83, 168]. Balasubramanian et al. and Kulkarni et al. as well as Zhao et al. suggested to reorganize the data transmission phases of the 3G protocol to save power. The goal of the works is to combine fragmented transmissions to longer but fewer transmission phases as each 3G transmission phase comes with an overhead in terms of an active power state.

2.2.2 4G Protocol

The successor of the 3G protocol is the 4G protocol that provides higher down- and upload data rates than its predecessor. Similar as for 3G, Huang et al. investigated the 4G protocol and developed a tool called *4GTest* [63]. They studied the effect of the Transmission Control Protocol (TCP) and the effect of application design on the network performance [64]. In [63], the group introduced 4GTest, a tool for the analysis of the performance characteristics and the power consumption of 4G. Using this tool, they found that although 4G has a higher throughout than 3G and even WiFi, the power consumption of 4G is much larger than the power consumption of the other two protocols. They identified a large tail energy as a key contributor to the energy consumption. Similar to some 3G power optimization techniques, they suggested to have fewer data transfers sending larger data chunks rather than many transfers sending small data chunks. Moreover, they found that the processing power can also be the bottleneck for web application when 4G is used. In [64], the group mainly investigated the TCP protocol and its effect on performance. They found that the TCP parameter settings, such as the TCP receive window, has a significant influence on the utilization of the protocol. Moreover, they showed that many network applications do not exploit the available bandwidth, which has a negative impact on the overall user experience and battery life. Similar observations were made by Nguyen et al. [107] and Li et al. [86], where [86] focused on the TCP performance in high speed trains. Furthermore, Xie et al. tackled this problem by introducing *CLAW*, a tool that boosts mobile web loading by taking into account information from the physical layer transport protocol [161].

2.2.3 WiFi Protocol

Besides 3G and 4G, the WiFi protocol is one of the most important protocols with respect to the wireless link power management. Hence, there exists a variety of work that targets different aspects of WiFi power management in mobile applications. Since WiFi is one of the most power hungry network interfaces on mobile devices, many works try to combine WiFi with other interfaces such as Bluetooth, 3G or 4G.

For example, Agarwal et al. presented an algorithm for reducing the energy of a phone for VoIP calls [1]. The energy consumption of the WiFi module is very high compared to the mobile data module for the HTC Tornado (Cingular 2125) smartphone used in the experiments. Hence, the group leveraged the data module to power on the WiFi module for incoming VoIP calls. This enabled a significant increase of the battery run time by a factor of 6.4 for the given setup. Manweiler et al. targeted a rather different approach for power saving [101]. Instead of implementing a power management algorithm on the client side, the authors presented a software modification of the access points for WiFi networks called *SleepWell*. The algorithm re-schedules WiFi packages on the client side (e.g., smartphone), which allows the clients to transfer packages sequentially rather than all at once. This minimizes waiting times for the clients and maximizes the WiFi sleeping time.

More works tried to optimize WiFi power consumption on the client side. For example, Chen et al. implemented a TCP packet re-scheduling algorithm using an additional buffer that prioritizes the packets according to the priority of the application [17]. Here, foreground applications have a higher priority than background tasks. Pefkianakis et al. tried to identify idle times of the WiFi network connection on smartphones based on the user activities [120]. If the user is not active, the WiFi module can enter a low-power state. Whenever activity is detected, the module is powered back on. Bandara et al. took a similar approach [9]. While the previous two works looked the idle time of the WiFi connection, Rattagan et al. looked at the situation where the WiFi module is used by multiple applications simultaneously [136]. To maximize the WiFi throughput, they falsified the WiFi status to background applications such that foreground applications could complete faster. This approach leads to 8 % power consumption reduction.

While the previously described works looked for a way to reduce the WiFi power consumption, [167] and [89] studied modeling and quantifying the power consumption of WiFi on mobile devices. Zhang et al. presented a model generator that involves several part of a smartphone, such as CPU, GPU, display, and also WiFi [167]. In [89], Li et al. looked at WiFi power consumption only and fine-tune the power model compared to [167].

2.2.4 Mixed Protocols

Many works tried not only to minimize the power consumption for a particular network protocol, but implemented a hybrid approach that mixes different wireless protocol types. Not only 3G, 4G and WiFi, but also Bluetooth is used in combination with other protocols because it consumes considerably less energy.

Pering et al. developed the tool *Coolspots*, inspired by the low energy consumption of Bluetooth [121]. Coolspots aims to minimize the energy consumption of a mobile device by balancing the transmissions between the WiFi and the Bluetooth interface. The challenge is to find an optimal configuration between the low bandwidth and range of Bluetooth and the higher range and bandwidth of WiFi, that comes with a much larger power consumption. This approach leads up to 50 % energy savings. Some similar approaches that tried to find the best configuration for WiFi and the mobile data interface - either 3G or 4G - were presented in various studies [133, 110, 49, 109, 58, 173]. The main difference of the works lies in the field of applications. For example, Nika et al. looked at general applications [109], while Zou et al. studied video streaming [173].

2.3 Display Power Management

When we aim to introduce hardware components that are contributing most to the power consumption of the smartphone for running applications, we cannot avoid looking into power management techniques for displays. As mentioned above, the majority of the power consumption of a mobile device can be accounted to the CPU, the wireless link and the display. Hence, in this section we will introduce power management techniques for the remaining one of these three components, the display.

Studies on display power management looked extensively at adjusting the illumination of the screen, especially for so-called LCD displays. Very briefly speaking, LCD displays use a crystal and filter electrodes in order to control a particular pixel. All pixels have to be illuminated by a backlight to produce the desired effect of forming a visible picture on the screen, which can vary in brightness. Many studies have looked into reducing the power consumption of the display by reducing the illumination of the backlight and adjusting the contrast to compensate for potential image quality loss [16, 22, 145, 103]. Iranli et al. suggested to take the intra-frame distortion component and the inter-frame distortion component, namely the spacial and the temporal components in a video frame into account to perform power management for LCD displays [73, 74]. Anand et al. implemented a gamma correction algorithm that enables significant power savings for LCD displays [6]. They verified their work on a laptop and two smartphones by performing a user study with 60 participants where they applied their approach to the game Quake III and could report power savings of up to 68 %.

Besides LCD screens, there is another promising technology that has been in the focus of industry and research community in the past years: OLED-based displays. The main difference between LCD and OLED technology is that each pixel of OLED displays can be illuminated by itself and no background illumination is necessary. This makes OLED displays thinner than LCD displays and the contrast of the display is usually more intensive as every pixel can be controlled individually. As the power consumption of an OLED display highly depends on the colors it shows, there were studies changing the colors of graphical user interfaces of different applications [38]. Dong et al. could reduced the power consumption of a commercial display up 75 % and also verified the acceptance of the change in a user study. Moreover, there were studies on DVFS-based approaches for OLED displays [146, 147]. Shin et al. scaled the supply voltage and adjusted the the pixels' R, G and B values based on a distortion factor. Wee et al. implemented an algorithm that dims areas of the OLED screen that are not relevant to a user to save power [157]. They performed a user study with 30 participants where they tested the algorithm on an Android device while playing the game Kwaak 3. The display power savings of 10 % seemed promising. The group generalized their approach in *Focus*, a frame work that applies the OLED power management strategy to a wider range of applications [152, 158]. They tested Focus on an Samsung Galaxy S3, studying 15 applications, among others Facebook, GMail and the Firefox browser and reported power savings between 23 % and 34 %.

Chen et al. presented another series of work on OLED DVFS [19, 20, 21]. They proposed a method where they scaled the voltage of small areas of different images by implementing a hardware driver [19]. To overcome quality loss caused by the downscaling, they remapped the colors such that it meets a certain Structural Similarity Index (SSIM). They reported power savings between 25.9 % and 43.1 %. The work was extended by the same group and applied to video streams [20, 21]. Similar to techniques for LCD displays, they applied spatial and temporal optimization on the supply voltage of the OLED display, leading to power savings between 19.05 % and 49.05 %.

Lin et al. reported that the Human Visual System (HVS) does not perceive all areas in an image with the same intensity [95]. This implies that changes or distortions in one part of the image might catch the attention of the user more than changes in another part of the image, or may not be perceived by the user at all. A technique called *image pixel scaling* is based on these findings [94]. While most techniques scale down complete areas of pixels at once, Lin et al. exploited the fact that every pixel in an OLED display can be tuned separately. While all regions of a picture are usually displayed at best quality, the group argued that only a few regions of a picture actually catch the full attention of a user. Hence, they identified those regions and displayed them in a good quality while other regions could tolerate more distortion to the image. The group reported power savings between 38 % and 42 % for tests on a Samsung Galaxy Tab 7.7. Another group around Anand et al. also exploited the HVS in their display power management approach [5]. The authors made use of the blue channel of the OLED display that consumes most power, but is least sensitive to the perception of the users. They generated colors that do not decrease the user experience, but consume less power. Moreover, they darkened areas of the image where the user pays least attention to. The group verified their approach in a user study and reported up to 45 % power savings. Further, Kim et al. proposed a form of content-related display power management [78]. They compared the contents of the display buffer with the frame that should be displayed next. Excluding the redundant frames, they could calculate the so-called content rate. Based on the content rate, the refresh rate of the screen could be adjusted.

3 Frame-based and Thread-based Power Management for Mobile Games

In this chapter, we look into game power management specifically for HMP platforms. There has already been fundamental work done on this topic by Dietrich et al. [33, 35, 36], but it has been conducted on an older hardware platform, that does not allow for the degrees of freedom in terms of power management that are possible on the big.LITTLE HMP platforms. Their hardware platform is a PandaBoard ES development board, that features only one dual-core CPU [114]. Hence, there is no option to shift tasks between different CPUs and there is no possibility to apply power-gating on a CPU level. Generally, games are very challenging applications in terms of power management because they usually contain a lot of unpredictable user interactions and are demanding in terms of computation power. Especially the interactive nature of games and the effect of power saving techniques on the user perception is an important problem. New power management techniques such as power gating or turning off CPU cores have to be explored in order to find the optimal balance between power and performance. This makes understanding how games work, their internal structure and their potential for power savings an important step in understanding Android applications in general.

3.1 Introduction

Games are one of the most favored, but also one of the most power consuming applications for mobile devices. It is reported that 34% of total mobile time is spent on gaming while 22% is spent on messaging and social networks [80]. The gaming workload is characterized as highly variable, and user-interactive as opposed to other types of mobile applications. These characteristics make it hard for the CPU frequency governor in an Android system to perform appropriate power management, thereby impairing the battery lifetime.

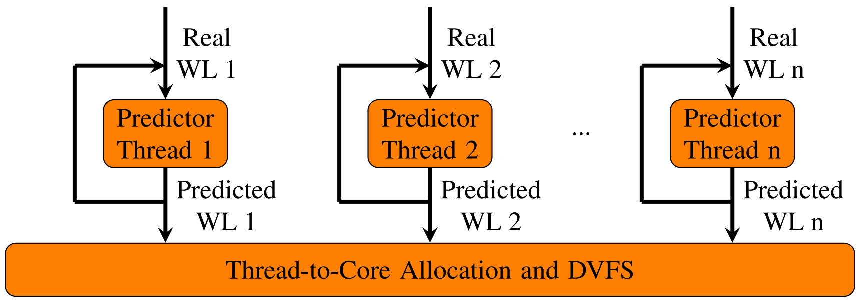


Figure 3.1: Power management unit with workload prediction on a per-thread basis, thread to core allocation and DVFS.

A fine-grained workload estimation has more potential for power reduction than coarse-grained workload estimations. Some works have focused on the short-term temporal correlation in computational workload in gaming and proposed frame-based workload predictors [36, 50, 52]. These works made fine-grained frame-wise workload estimations to perform DVFS, and hence, minimize the power consumption while not violating the performance requirements. The performance requirements of games are usually defined by FPS. The user perceived quality is not affected as long as the FPS value stays above a certain threshold, e.g., 30 FPS [27]. Yet, the frame-wise prediction of gaming workloads is a demanding task, as the workload characteristics differ among games and underlying hardware platforms.

To meet the dynamic computing demands, HMP SoCs are being embedded in state-of-the-art mobile devices. They comprise a number of heterogeneous cores to allow computationally demanding threads to run on the performance-oriented cores, and less demanding threads to run on the energy-efficient cores to save energy. This way, the system is able to respond to various computing demands in an energy-efficient manner. However, the problem of minimizing the power consumption of games by means of thread allocation to cores and DVFS without significant degradation in user perception is a demanding task. Previous works [36, 50] have looked into gaming workloads as a whole, and performed power management for the entire application. Unlike in these works, multiple threads of one game could be distributed over multiple cores, enabling the cores to run on the minimum frequency that just fulfills the FPS requirement of the user. Consequently, it is essential that the gaming workload is estimated on a per-thread and per-frame basis as shown in Figure 3.1. We observed that there are roughly two categories of thread workloads, periodic and aperiodic. Hence, we propose to make use of a hybrid predictor, a combination of the Autocorrelation Predictor (ACR), and the Weighted Moving Average (WMA), to handle different workload categories appropriately. However, even if we know the exact values of workload, finding the energy-optimal allocation and frequency is known to be computationally intractable [7]. Thus, a heuristic algorithm has to be developed to perform the actual power management.

In this work, we extend the previous frame-based power management techniques to HMP platforms to propose a frame-based and thread-based, predictive power manager for mobile games. The proposed framework is capable of performing thread allocation and DVFS simultaneously to meet the FPS requirement of the user while minimizing the power consumption. The contributions of this chapter are summarized as follows:

- We characterized the gaming thread workloads and develop a thread-based and frame-based hybrid workload predictor to accurately predict the gaming workload.
- The hybrid predictor learns online whether the thread workload is periodic or aperiodic.
- We devised an algorithm to perform thread-to-core allocation and DVFS simultaneously based on the predictions.
- We implemented the integrated power manager, which we refer to as *GameOptimized governor* in the following, on an Odroid-XU3 development board [59] and compared it to the default Android governors.
- We performed a user study to evaluate the user perception of our proposed power manager.

The experimental results show that up to 60.0% of energy can be saved for our power manager, while the user perception is considered good.

3.2 Related Work

Power management for mobile games has drawn attention rather recently as opposed to techniques for other multimedia applications such as videos. Prior works regarding fine-grained, i.e., per-frame, power management of mobile gaming is scarce due to the difficulty of precise workload prediction and closed-source nature of commercial games. A PID controller-based workload predictor has been proposed for 3D games, but it requires parameter hand-tuning and is subject to trade-off between the controller stability and prediction accuracy [37]. Despite the hand-tuning, the PID predictor diverges for different states of the game, e.g., the loading and the gaming phase as those two have significantly different workload behavior. To overcome this issue, auto-regressive (AR) and self-tuning Linear Mean Square (LMS) predictors have been proposed [36, 35], which have shown reasonable accuracy in workload prediction. These predictors estimate the workload per frame to perform DVFS in order to reduce power consumption. However, these predictors are only able to predict the frame workload as a whole, while per-thread prediction is required to perform thread-to-core allocation on HMP platforms. Moreover, these predictors are not able to distinguish between different types of thread workloads, e.g., periodic and aperiodic. Hence, their prediction accuracy is not guaranteed for all types of threads.

Power management utilizing both thread-to-core allocation and DVFS on an HMP platform was proposed in [118]. The work introduces a heuristic online strategy based on a *thread-price* calculated from the CPU utilization. It allocates the threads to cores in a way that allows for reducing the CPU frequency, and hence, the power consumption. Another work from the same group has proposed a coordinated CPU-GPU power management that could perform better than independent management for 3D games [117]. This work has been extended to use a regression-based predictor for the impact of DVFS on the game workload in [116]. Although these lines of works achieve decent power reduction compared to the default Android governor, the power management is mostly done in an inherently reactive way to workload changes, and applied in a coarse-grained manner, i.e., roughly once per-second. Compared to these works, our proposed power management offers more potential for power reduction as we are able to better exploit the per-frame workload changes by the predictive nature of our approach.

In [23], the authors introduce a game state and frame rate dependent DVFS policy. The work observes that there exists a bottleneck CPU frequency above which the frame rate does not increase anymore. Moreover, it changes the target frame rate itself based on the game state detection. The CPU frequency is scaled such that it meets the target frame rate.

In summary, we propose a predictive frame- and thread-based CPU power management scheme for games on HMP platforms, which integrates scheduling and DVFS. Compared to previous work, we identify the workload type of each thread, e.g., periodic or aperiodic, and apply the most suitable predictor for that thread. Based on the predicted workload of threads, we allocate the threads to the CPU cores such that the workload is evenly distributed over the cores at the minimum required CPU frequency to meet the target FPS. Rather than relying on the average FPS as a sole performance metric, we also perform a user study to evaluate our power manager.

3.3 Workload Characterization of Mobile Games

Previous work proved that frame-based workload prediction and DVFS for games achieves significant power savings [36, 37]. Game scene changes, e.g., the appearance of a new enemy, usually occur suddenly and last a number of seconds. Hence, subsequent game frames inherit similar workload, which can be exploited for workload prediction-based power management. Also, mobile games are becoming more and more multi-threaded. Therefore, it is essential to understand the per-frame and per-thread workload characteristics of games to perform the proposed fine-grained power management.

We observe that there are two types of threads in a typical gaming process, periodic and aperiodic. Figures 3.2 and 3.3 show the workload over time for the two categories of threads. The workload was measured on an HMP platform based on the Exynos5422 processor running an Android operating system that will be described later. Figure 3.2 depicts a sample thread workload of the game Asphalt. The workload is aperiodic, but shows high temporal correlation among adjacent frames. Figure 3.3 shows the workload of a periodic thread in the game Grand Theft Auto (GTA) 3. There is not much temporal correlation among adjacent frames, but the workload is invoked every second. Once we understand the pattern, it becomes easier to predict the thread workload. The AR predictor used in [36] performs decently for the aperiodic category of threads, but it is not so efficient in predicting the periodic threads. This observation is the key to our proposed hybrid WMA predictor that successfully estimates workload for both categories of threads.

Another thing to note is the performance requirement of games, usually measured in FPS. Most modern games target a frame rate of 60 FPS to guarantee a good gaming experience for the

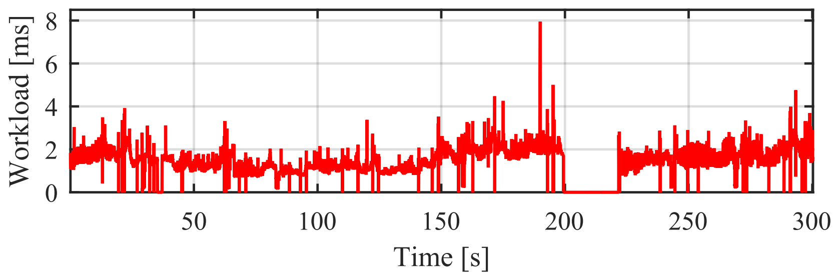


Figure 3.2: Measured non-periodic workload of one thread from the game Asphalt.

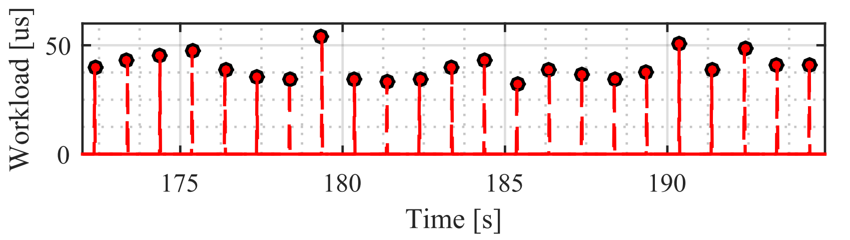


Figure 3.3: Measured thread workload from the game GTA 3 which is reoccurring periodically. The workload is zero except for the indicated circles.

user. This means that a game frame is computed within 16.7 ms. Research has shown that lower frame rates of 30 FPS are barely noticeable for the user in shooter games [27]. Overachieving this goal requires a great amount of computational resources, and hence, battery energy, which is not desired in the case of mobile devices. Some game developers have already taken this into account and run their games with only 30 FPS. We also adopt this result and target to achieve 30 FPS within our GameOptimized Governor.

In general, the GameOptimized governor is designed for applications that exhibit similar workload characteristics for subsequent frames. It learns the workload within a range of frames and can be applied to all applications that maintain a constant frame rate by initiating periodic frame buffer changes within the OpenGL library as explained in Section 3.5.3.

3.4 Frame-based and Thread-based Workload Prediction

Our GameOptimized governor relies on a precise workload prediction of each thread as it performs fine-grained per-thread and per-frame power management. Overestimation of the thread workload will hinder the power saving potentials as it forces the cores to run at a higher frequency, while underestimation will degrade the user experience by violating the FPS constraint. The criteria for choosing the predictor are 1) the applicability for a wide range of games without the need of manual parameter tuning for every single game and 2) the applicability for the various types of thread workloads we have discussed in Section 3.3.

We have compared a set of predictors that have been previously considered for estimating the per-frame gaming workloads, and a number of hybrid predictors. The predictors are the PID, the simple moving average (SMA) [149], WMA [149], linear least squares (LLS) [60], LMS [60], and the ACR [150] predictors. Moreover, we evaluate three hybrid predictors that combine the ACR predictor with the PID, the SMA and WMA predictors. We have not considered the AR predictors used in previous works as they are not suitable for thread-based workload prediction. AR predictors need to be trained with recorded workloads to achieve a good result. As there can be hundreds of game threads that exhibit very different workloads, it is neither feasible to obtain the optimal weights for each thread nor to obtain one set of weights for all threads. The chosen predictors are fed with measured thread workloads of five popular games, Dragon Fly, Star Wars Galactic Defense, Blood and Glory: Legend, GTA 3, and Asphalt 8, from different genres that exhibit different workload characteristics.

Table 3.1 shows the accuracy of each predictor for each game. The results per game are shown in terms of error of prediction, err_{game}, which is defined as follows.

$$err_{game} = \frac{1}{M} \sum_{\forall tid} \frac{1}{N} \sum_{\forall n} |W_{tid,m}(n) - W_{tid,p}(n)|, \tag{3.1}$$

where $W_{tid,m}[t]$ and $W_{tid,p}[t]$ are the measured workload and predicted workload of thread tid at time slot n, M is the number of threads in the game, and N is the total number of frames within the measurement period.

Table 3.1: Averaged prediction error e_p in percent.

Predictor	Asph. 8	Drag. Fly	Glad.	GTA III	Star W.
PID	9.74	40.97	31.10	3.51	10.45
SMA	9.18	38.32	28.76	3.47	10.17
WMA	9.03	38.37	28.78	3.39	10.04
LLS	9.87	44.46	31.77	3.64	10.63
LMS	unstable	unstable	unstable	unstable	unstable
ACR	9.73	552.74	34.68	2.90	11.56
Hyb. PID	7.79	39.34	29.44	2.82	10.00
Hyb. SMA	7.51	37.35	27.72	2.82	9.67
Hyb. WMA	7.41	37.36	27.66	2.80	9.61

Our results show that the hybrid predictor that combines the autocorrelation and the WMA predictor has the smallest prediction error. Hence, we have chosen this predictor and implemented it within our GameOptimized governor. While the ACR predictor performs well for periodic workloads, the WMA predictor achieves good results for aperiodic workloads. However, applying either only the WMA or the ACR predictor to all of the threads results in sacrificing prediction accuracy for the periodic or aperiodic thread workloads, respectively. This can also be seen from Table 3.1, where the hybrid predictors have a lower prediction error than both comprising predictors separately. In the subsequent sections, we first describe the two comprising predictors, the WMA and ACR predictors before we describe the proposed hybrid WMA predictor.

3.4.1 Weighted Moving Average Predictor

The WMA predictor is motivated by weighted moving average filters [60]. The thread workload W of a frame is estimated by calculating the weighted average of the past N frames,

$$W(n+1) = \frac{\sum\limits_{i=0}^{N-1} (N-i) \cdot W(n-i)}{\sum\limits_{i=1}^{N} i}, \tag{3.2}$$

where $W(n)$ is the workload of the n-th frame in seconds. The weights are chosen to be larger for more recent data such that it has greater influence on the prediction than older data. The window size, N, can be tuned. Inspection of different values for N showed that $N = 14$ leads to a good prediction for the WMA. Figure 3.4 shows an example workload from the game Asphalt and the corresponding prediction of this signal by the WMA predictor. As shown in the figure, the predicted curve follows the mean of the highly fluctuating workload.

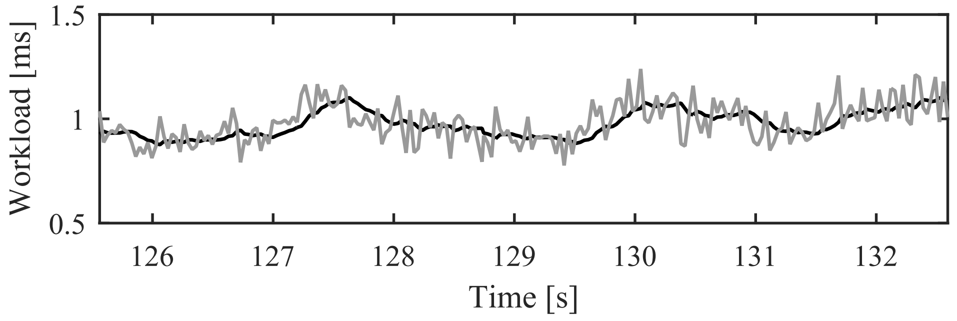


Figure 3.4: Workload prediction with the WMA Predictor. The black line represents the predicted workload while the gray line shows the original signal.

3.4.2 Autocorrelation Predictor

As stated in Section 3.3, some thread workloads show periodic behavior. Especially for these particular threads whose workload is zero most of the time, the WMA predictor does not result in satisfying prediction accuracy. However, such workloads are highly correlated to a shifted version of themselves. In other words, the workloads exhibit a high autocorrelation, which can be exploited in order to achieve better prediction. The ACR predictor is capable of exploiting the repetitions within a thread workload. The autocorrelation $ACorr(W, \tau)$ of the workload W at lag τ

$$ACorr(W, \tau) = \frac{ACovar(W, \tau)}{ACovar(W, 0)}, \tag{3.3}$$

is obtained by normalizing the autocovariance $ACovar(W, \tau)$ given by [150],

$$ACovar(W, \tau) = \sum_{i=0}^{N-\tau-1} (W(i+\tau) - \overline{W}) \cdot (W(i) - \overline{W}), \tag{3.4}$$

where $\overline{W}$ is the arithmetic mean of the workloads W. If $abs(ACorr(W, \tau))$ is close to 1, the workload W is highly correlated with itself, shifted by τ samples. By calculating the autocorrelation for $\tau = \{1, 2, ..., \Upsilon\}$, the lag with the highest autocorrelation τ_{Max} can be identified. The new workload is then estimated as

$$W(n + 1) = W(n - \tau_{Max}). \tag{3.5}$$

Using this predictor, a perfectly periodic signal can be precisely predicted after a certain settling time. The total time it takes to learn the signal depends on the period of the signal. A perfectly periodic signal was created to test the autocorrelation predictor. Figure 3.5 shows an example in which it takes two periods for the ACR predictor to learn the periodic signal.

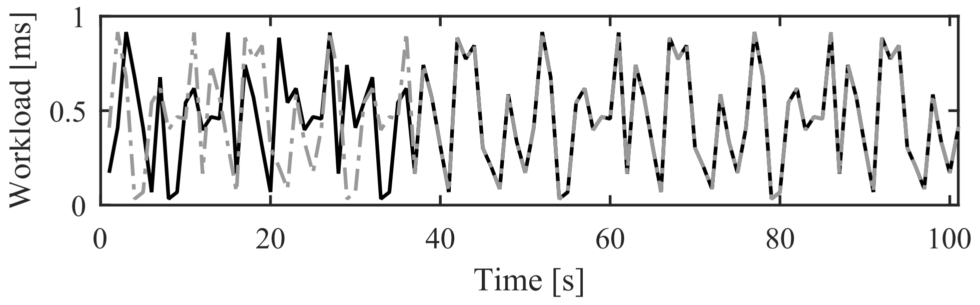


Figure 3.5: Workload prediction with the ACR Predictor on a test data set with perfect repetitions. The solid line represents the predicted workload while the dashed line shows the original signal.

3.4.3 Hybrid WMA

The ACR predictor performs well if the predicted signal is highly autocorrelated. However, the predictor is not applicable for signals with low autocorrelation. For this reason, the ACR predictor was combined with the WMA predictor. After calculating the autocorrelation at lags τ for $\tau = \{1, 2, ..., \Upsilon\}$, the absolute maximum of the obtained autocorrelations is considered. If this maximum is above a threshold Θ, the prediction is done using the ACR predictor, otherwise the WMA predictor is used.

For the hybrid WMA predictor, we need to tune the parameters Υ, N, and Θ. Υ is the maximum number of lags to find whether the signal exhibits autocorrelation or not. N is the number of averaged data points for the WMA predictor. Θ is the threshold that indicates whether the ACR or the WMA predictor should be used. We have estimated the parameters experimentally by applying the predictor with varying parameter sets to the workload of the five games mentioned above. Then, we chose the results with the least prediction error and calculated their averages. In the following, we exemplify how we have determined the parameter N. For each game and thread, we have predicted the workload using the WMA predictor varying the parameter N $= \{1, 2, ..., 100\}$. For each run, we calculated the workload prediction error $err_{game,N}$. Then, we determined the minimum $err_{game,min}$ for each game and averaged all minimum errors to obtain the resulting parameter N.

The approaches for Υ and Θ are similar, so we will not elaborate on this due to space reasons. Finally, the resulting values are N $= 14$, $\Upsilon = 20$ and $\Theta = 0.34$.

3.5 Game Power Management

In this section, we provide an overview of our proposed game power management based on the predictor we have stated above. First, we provide an overview of the complete framework. Next, we explain the underlying HMP platform and software components we have implemented. Finally, we present a heuristic algorithm that allocates threads to CPU cores and performs DVFS based on the workload prediction.

3.5.1 Overview of Frame- and Thread-based Power Management

The proposed frame-based and thread-based power management execution flow is summarized as follows:

1. Detect whether the running application is a game or not
2. Detect the beginning of a new frame
3. Predict the workload of each running thread
4. Allocate the threads to CPU cores according to the predicted workload
5. Set the CPU frequency to meet the desired frame rate

The proposed power management policy is implemented mainly in the Android governor. First, the governor needs to distinguish between the game processes and other application processes as we propose a power management technique tailored for mobile games. This is done by maintaining a hash table of known games. Second, the governor is notified about the beginning of a new frame. As our whole framework is based on per-frame power management and accurate prediction of the thread workloads, it is important to detect the precise point in time when the processing of a new frame begins. This is accomplished by modifying the OpenGL library, as described below. Third, we predict the workload of each thread required to process the frame. We implement the predictor described in Section 3.4. Fourth, we apply a heuristic algorithm to allocate threads to each core and set the operating frequency as will be described in Section 3.5.4. The basic idea of the algorithm is to prefer the A7 CPU and distribute the threads' workloads as evenly as possible among all cores as long as the FPS constraint is not violated.

3.5.2 HMP Hardware Platform

Although we have already introduced the underlying hardware platform in Section 1.1.2, we want to give a short recap of the most important hardware features of the Odroid-XU3 board. It features an Exynos5422 MPSoC that is also part of the Samsung Galaxy S5 smartphone [59].

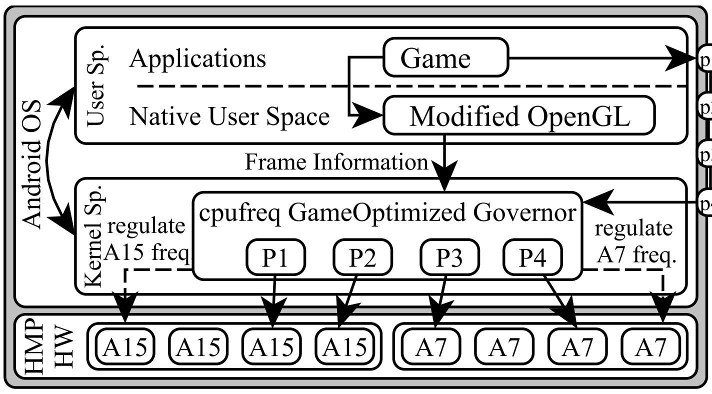


Figure 3.6: System architecture with modified OpenGL library and combined governor/scheduler unit within the GameOptimized governor.

This heterogeneous MPSoC is based on the ARM big.LITTLE architecture, consisting of two different CPU clusters. One is a performance-optimized Cortex-A15 quad-core CPU (A15) and the other one is a power-optimized Cortex-A7 quad-core CPU (A7). We refer to a CPU cluster with a set of CPU cores as a *CPU* while we refer to a single CPU core of a CPU cluster as a *core*. The CPU voltage and frequency can be adjusted independently per CPU but not per individual core. The A7 supports a frequency range from 1.0, 1.1,$\cdots$ to 1.4 GHz while the A15 supports a range from 1.2, 1.3,$\cdots$ to 2.0 GHz. In addition, a Mali-T628 GPU and 2 GB of LPDDR3 memory are integrated into the Exynos5422 SoC. The operating system distribution of our setup is an Android Kitkat 4.4.4 with a Linux kernel version 3.10.9. The setup features so called *HMP scheduling* that allows for the allocation of tasks to all big and little cores simultaneously. The HMP scheduler prefers the small cores, and only migrates threads to the big cores if the CPU utilization rises above a certain threshold.

3.5.3 Software Architecture

As mentioned before, our power management technique is implemented as an Android CPU governor. Figure 3.6 shows the relevant software entities, which are the *GameOptimized governor*, and the *modified OpenGL library*.

GameOptimized Governor

The GameOptimized governor, the key part of our power management technique, performs workload prediction, thread allocation and DVFS. It is implemented as a Linux kernel governor, which is a part of the *cpufreq* module. It contains a set of workload predictors that predict the workload for each game thread.

Modified OpenGL Library

The GameOptimized governor needs to be aware of the game context such as whether a game is being played, and the start time of the frame rendering. This is achieved with the help of the modified OpenGL library. The Android games we look into are downloaded from the Google Play Store. They are closed-source, and hence, we cannot instrument the source code to detect when a frame has been processed. Similar to the approach in [35], we calculate the frame rate by modifying the *eglSwapBuffers()* function of the OpenGL library. Usually, the frame buffer of a graphics unit consists of two buffers, a front and a back buffer. The currently displayed frame is stored in the front buffer while the next image is rendered into the back buffer. The function eglSwapBuffers() swaps the two buffers and the new frame is shown on the display. After this call, the processing of the next frame begins. We calculate the frame rate by measuring the time between two such function calls. This information is passed to the GameOptimized governor via an *ioctl()* call.

Game detection is also done in the modified OpenGL library. The process ID of the calling process is determined using the system's *getpid*() function. Via the process ID, the name of the process is read from the kernel's /proc file system. Next, a hash function is used to identify the process. All hashes of known games are calculated once and stored in a list.

3.5.4 Thread Workload Prediction-Based Core Allocation and Frequency Selection

In this section, we present our strategy for the thread-to-core allocation and DVFS as well as the underlying hardware and software models. For the models, we take a cue from [118].

Hardware Model

Our hardware platform comprises of two CPUs P_x where $x \in \{big, little\}$ with different performance characteristics, described in Section 3.5.2. Each P_x incorporates N_x cores where each core is denoted as $P_{x,i}$ with $i = 1, 2, ..., N_x$. The cores operate at the frequency f_x that ranges from $f_{x,min}$ to $f_{x,max}$. Each core of each CPU provides a maximum capacity $C_{x,i}$. The capacity is defined as the number of CPU cycles that can be executed on one CPU core $P_{x,i}$ at the frequency $f_{x,j}$ within a given time. The time is dependent on the target frame rate FR_T. As introduced in [118], we define a *Migration Factor* that represents the performance difference between the big and the little cores. While one core of P_{little} can execute a number of n instructions, one core of P_{big} can execute $MigrationFactor_{big} \cdot n$ instructions within the same time. Experimentally, we have found that $MigrationFactor_{big} = 1.7058$ is a suitable number for the given platform. $MigrationFactor_{little}$ is normalized to 1. Hence, the maximum capacity of one core is calculated as

$$C_{x,i,max} = MigrationFactor_x \cdot \frac{f_{x,j}}{FR_T}, \tag{3.6}$$

while $C_{x,i}$ is the currently available capacity of one core and FR is the target frame rate. It needs to be considered that $C_{x,i}$ is further influenced by other threads executing on the CPUs, for example, threads spawned by the operating system. Therefore, we add an integral controller I_t to the calculation of the maximum available capacity for the game threads with

$$I_t = \sum_{1}^{n} \alpha \cdot (t_F - t_T), \tag{3.7}$$

where t_T is the target time for a frame to compute, t_F is the actual computation time, n is the total number of computed frames and α the integral gain of the controller. As α highly

Table 3.2: Power consumption of the A15 at a utilization of approx. 70%.

Volt.	Freq.	Idle	1 Core	2 Cores	3 Cores	4 Cores
1.0 V	1.2 GHz	0.30 W	0.70 W	1.07 W	1.46 W	1.80 W
1.0 V	1.3 GHz	0.34 W	0.79 W	1.21 W	1.65 W	2.05 W
1.0 V	1.4 GHz	0.37 W	0.85 W	1.35 W	1.80 W	2.26 W
1.0 V	1.5 GHz	0.42 W	0.98 W	1.51 W	2.03 W	2.57 W
1.1 V	1.6 GHz	0.48 W	1.13 W	1.76 W	2.37 W	2.99 W
1.1 V	1.7 GHz	0.55 W	1.30 W	2.01 W	2.74 W	3.50 W
1.1 V	1.8 GHz	0.66 W	1.48 W	2.30 W	3.20 W	4.05 W
1.2 V	1.9 GHz	0.73 W	1.72 W	2.74 W	3.85 W	4.92 W
1.3 V	2.0 GHz	0.93 W	2.23 W	3.49 W	4.93 W	6.49 W

influences the speed at which the current CPU frequency adopts to frame misses, we introduce an α_{up} and an α_{down}. Experimentally, α_{up} was tuned for a too low frame rate, while α_{down} was tuned for a too high frame rate. We found that $\alpha_{up} = 0.2$ and $\alpha_{down} = 0.1$ are suitable for our application. Finally, I_t is converted to the actual control value I_c, which is measured in CPU cycles, and subtracted from the maximum available capacity $C_{x,i,max}$.

Software Model

A game consists of a number N_t threads T_k where $k = 1, 2, \cdots, N_t$. Each thread puts a workload W_k on the system that is measured in CPU cycles. The workload W_k of each thread takes up capacity on one CPU core $P_{x,i}$.

Power Consumption Characteristics of the Hardware

As described in Section 3.5.2, our hardware platform comprises the power-efficient A7 CPU and the performance-oriented A15 CPU. Our measurements have shown that the A7 consumes significantly less energy than the A15 for the same workload, although the A15 computation time is significantly lower. Hence, it is more efficient in terms of energy consumption to shift as much workload as possible to the A7 and only switch to the A15 when the required FPS cannot be met. However, it is not energy-efficient to process the workload at the maximum frequency as it results in higher energy consumption.

Furthermore, we have run a measurement set, which shows that it is more power-efficient on our platform to distribute the threads over all available A15 cores and lower the frequency rather than utilizing fewer cores at a higher frequency. This is mainly due to the impossibility of turning off single A15 cores and the resulting high idle leakage currents of these cores. Table 3.2 shows the power consumption of the A15 at different frequency levels and different numbers of utilized cores. The load of each utilized core is approximately 70%. Running one core at 1.9 GHz (case 1) and running four cores at 1.2 GHz (case 2) consumes approximately the same amount of power. However, if we compare the workload of both cases to the workload of one core at 1.2 GHz, the workload of case 1 is only 58% larger, while the workload of case 2 is 300% larger. Hence, in this extreme case, we can execute 5 times the amount of work on multiple cores in case 2 consuming the same amount of power as in case 1. Based on these observations, we implement a strategy that aims to prefer the A7 if the FPS requirement is not violated. Further, we distribute the workload as evenly amongst the cores as possible to keep the CPU frequency as low as possible.

Thread to Core Allocation

The strategy we use to distribute a game's tasks to the CPU cores is shown in Algorithm 3.1. It is executed once every frame. First, frequencies of both CPUs P_x are set to the minimum value and the capacity $C_{x,i}$ of each core is reset. Then, the workload of the next frame is predicted for each game thread. Next, we iterate through the threads T_k and assign them to CPU cores. We begin with the A7 cores at the minimum frequency level $f_{little,min}$. To prevent re-allocation overhead, we first check whether the previously assigned core offers enough capacity for the current thread. If not, we assign it to the core with the most available capacity. However, if none of the cores at the current frequency level provides enough capacity $C_{x,i}$ for the workload W_k of thread T_k, we increase f_{little} to the next higher frequency level. If we cannot find a suitable core on the A7 CPU, we reset its frequency to the previous level. Then, we repeat the same procedure for the cores of the A15 CPU. If we cannot find a suitable core on the A15, we assign the thread to the A15 core with the minimum workload.

3.6 Evaluation and User Study

We have evaluated the GameOptimized governor by comparing it to the two most popular Android default governors, interactive and ondemand. Moreover, we performed a user study to test the users' perception of the implemented governor. The results show that we save on average 41.9% of energy compared to the interactive and 31.2% compared to the ondemand governor with only a small degree of perceptible performance loss.

3.6.1 Energy Consumption and Frame Rate Evaluation

The main objective of the GameOptimized governor is to lower the power consumption of the two CPUs on the Odroid-XU3 board during the game play. To evaluate the governor, we chose twelve games of different genres. We played each game three rounds using 1) our GameOptimized governor, 2) the interactive governor, and 3) the ondemand governor. Each round had a duration of 10 minutes. For comparable results and synchronized measurements the board was rebooted before each measurement. To avoid that changes of the GPU frequency interfere with the frame rate, the GPU frequency was fixed to the maximum possible value of 543 MHz.

Figure 3.7 shows the total CPU energy consumption (A7 and A15) for the twelve test games for each of the three governors. The GameOptimized governor can achieve noticeable energy savings for all of the games, up to 60.0% compared to the interactive governor and 58.5% compared to the ondemand governor. In general, the savings highly depend on the type of the game and hence, the workload that is generated. Games like I, Gladiator, Fruit Ninja, Dragon Fly and Sonic Jump do not generate high workloads. Consequently, both default governors whose power management strategy is highly workload dependent, do not ramp up the frequency of the A15 as they do for example for GTA 3 or Interstellar. For those less resource demanding games, the energy consumption using the default governors is anyway comparatively small.

```
1: Initialize $f_x = f_{x,min}$ for $x \in \{big, little\}$
2: $C_{x,i} = C_{x,i,max} - I_c$ at $f_{x,min}$ and $i = 1..N_x$
3: Predict workload $W_k$ for each thread $T_k$ where $k = 1..N_t$
4: for $k = 1$ to $N_t$ do
5:     if $W_k == 0$ then
6:         Do not change allocation of $T_k$
7:     else
8:         Set $x = little$
9:         while $f_x <= f_{x,max}$ do
10:            Check previous allocation and if needed iterate
11:            through all cores at $f_x$ to find core with max($C_{x,i}$)
12:            if max($C_{x,i}$) $> W_k$ then
13:                Allocate $T_k$ to $C_{x,i}$
14:                $C_{x,i} = C_{x,i} - W_k$
15:                Continue with next thread $T_k$
16:            else
17:                Increase $f_x$ to the next level
18:            end if
19:            if $f_{little} == f_{little,max}$ then
20:                Set $f_{little}$ to previous value and restart
21:                while-iteration with $x = big$
22:            end if
23:            if $f_{big} == f_{big,max}$ then
24:                Allocate $T_k$ to $P_{big,i}$, with $max(C_{big,i})$
25:                Continue with next thread $T_k$
26:            end if
27:        end while
28:    end if
29: end for
```

Algorithm 3.1: Thread to core allocation and frequency selection strategy.

Hence, the power savings for the GameOptimized governor are smaller than for highly resource demanding games. Figure 3.8 shows the average FPS for the twelve test games for each of the three governors. As described in Section 3.5.1, our implementation is designed in a way that it targets a frame rate of 30 FPS to guarantee a good user experience. The figure shows that the average frame rate for all games is between 30 and 40 FPS. For Dragon Fly and Sonic Jump, we can observe that the achieved frame rate is higher than for the other games, approximately 50 FPS. This effect is caused by the above mentioned low resource demand of those games. Even by applying the GameOptimized governor's thread allocation scheme and frequency down scaling, one frame is processed faster than within 33 ms. Consequently, the frame rate clamps at a higher value. A solution to this problem is to apply a more aggressive power management technique, for example delay the call to the eglSwapBuffers() function. Another effect that becomes visible in Figure 3.8 is the frame limitation, which is by default incorporated in some of the games. For Interstellar and Asphalt 8, the frame rate is already limited to 30 FPS. The game

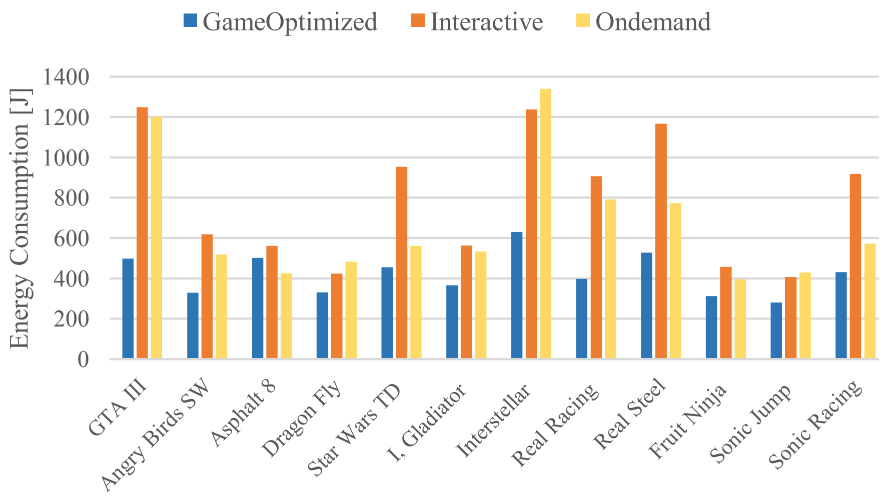


Figure 3.7: Total energy consumption of both CPUs for the twelve test games for the game power manager and the Android default governors.

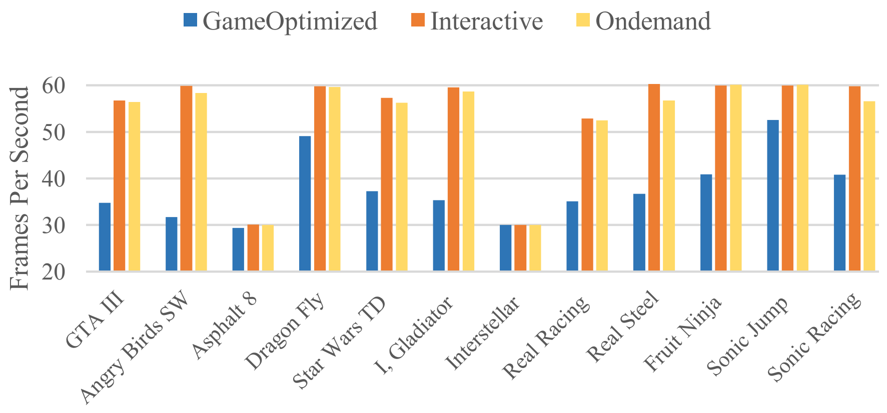


Figure 3.8: Average frame rate for the twelve test games for the game power manager and the Android default governors.

Interstellar is a good example for showing the energy saving efficiency of the GameOptimized governor. Although the frame rate stays the same as for the other two governors, the energy consumption is halved. Hence, we can claim that the energy savings we achieve are not only due to the frame rate reduction, but because of the applied thread to core allocation and DVFS scheme.

3.6.2 User Study

In the previous section, we have shown that that the GameOptimized governor achieves high energy savings. Due to the reduced target frame rate of 30 FPS, it needs to be ascertained whether a decrease in the user experience is perceivable or not. For this reason, a user study with ten participants was conducted. The goal of this study was to obtain a realistic rating of the gaming performance of the GameOptimized governor compared to the Android default interactive governor. For the study, we have chosen five games, a subset of the twelve games presented in Section 3.6.1: GTA 3, Angry Birds Star Wars, Interstellar, Fruit Ninja and Sonic Racing. The reduction of the game number from twelve to five leads to a duration of the study of about one hour per person.

For each game, there are two phases, the training phase and the actual playing phase. During the training phase, the participant can try the game for an unlimited amount of time to get used to the game play and the controls. In the playing phase, the participant plays every game twice for three minutes, once with the GameOptimized governor (measurement 1) and once with the interactive governor (measurement 2). The governors are picked in random order, hence, the user does not know which governor is currently active. The Odroid-XU3 board is rebooted before each measurement. After one measurement, the user is asked to rate the game play. The possible grades are in a range from one to six, where one is the best (no lags or glitches) and six is the worst (game is not playable).

Table 3.3 shows the rating results from all participants for all games. We can see that a small performance decrease was notable for the GameOptimized governor for four of the five games. Especially Fruit Ninja, which is a highly interactive game with a lot of fast animations, was rated worse than the other games. Still, the participants considered the gaming experience as *good* and *very good* for most of the measurements. Moreover, the grading difference between measurement 1 and measurement 2 is not more than 1 step apart for almost all cases. After the measurements, we showed the participants the amount of energy (in percent) that could be saved by applying the GameOptimized governor. All of them agreed to compromise a little performance for the energy savings that can be achieved with the GameOptimized governor.

Table 3.3: Gradings in the user study for the GameOptimized governor (G) and the interactive governor (I) per participant (P).

Game	GTA III		Angry B.		Interstel.		Fruit N.		Sonic R.	
Gov.	G	I	G	I	G	I	G	I	G	I
P1	2	1	2	1	1	1	2	1	1	1
P2	1	1	2	1	1	1	2	2	1	1
P3	2	2	1	1	1	1	1	1	1	1
P4	1	1	2	1	2	2	1	1	1	1
P5	1	1	1	1	1	1	1	1	1	1
P6	2	2	2	3	1	1	4	3	2	2
P7	1	2	1	1	1	1	1	1	1	1
P8	1	1	2	1	1	1	2	1	1	1
P9	2	1	2	2	1	1	2	1	2	2
P10	2	1	1	1	1	1	3	1	3	1
∅	1.5	1.3	1.6	1.3	1.1	1.1	1.9	1.3	1.4	1.2

3.7 Summary

In this chapter, we have presented a predictive, thread- and frame-based Android game power manager on an HMP SoC, the Odroid-XU3 board. Compared to previous works that have looked either into workload prediction or thread-to-core-allocation, we combine the advantages of a frame-based workload predictor with an energy-aware thread-to-core-allocation technique on the power-efficient little CPU and the performance-oriented big CPU. We predict the workloads of all game threads per frame and use this information to distribute the threads over the CPU cores such that we can minimize the CPU frequency, and hence, save energy. The predictor differentiates between periodic and aperiodic workloads. We evaluate our power manager in a user study, which reveals that our power manager can save on average 41.9% of total energy consumption while still maintaining a *good* and *very good* user experience.

This work is an important step towards showing that applying power management strategies on HMP platforms can lead to very high power savings without impacting the user performance dramatically. We have shown that especially preferring the little cores where possible achieves high saving rates. Moreover, reducing the frequency of the big cores to meet the demand of the application proved efficient. Hence, it is important to further look into interactive applications and find a balance between power and performance.

4 Web Browser Workload Characterization for Power Management

In this chapter, we evaluate a new application type in respect to its properties and its power consumption on HMP platforms: The web browser. While games are mostly interactive, which is challenging for power management, the browser exhibits multiple different states that are also common for other Android applications. For example, it can playback videos, features scrolling through different web pages and, most important, downloads and renders different types of contents from the web. Loading web pages is a very expensive task in terms of power consumption. Hence, the potential of saving power is very large for this task. To identify how much power we can save for loading a web page on an HMP platform, we tweak the different parameters that are available on the platform – especially on the big cores – and present a detailed analysis and characterization of the power versus performance trade-off for web browsers on HMP platforms.

4.1 Introduction

The number of mobile users has increased rapidly over the past few years and is reported to surpass desktop web browsing traffic. Google reports that already more searches take place on mobile devices than on desktops in ten major countries, including the US and Japan [48]. This trend is likely to accelerate with the exploding sales of tablet devices which has grown ever faster than PCs [97]. Not only mobile web traffic, but also the computational demand of the mobile web pages is increasing significantly [171].

Mobile web browsing is enabled by mobile browsers such as Chrome, Safari, etc. A browser consists of multiple components such as the user interface, browser engine, layout engine, display components, and networking. The most time and power consuming component while

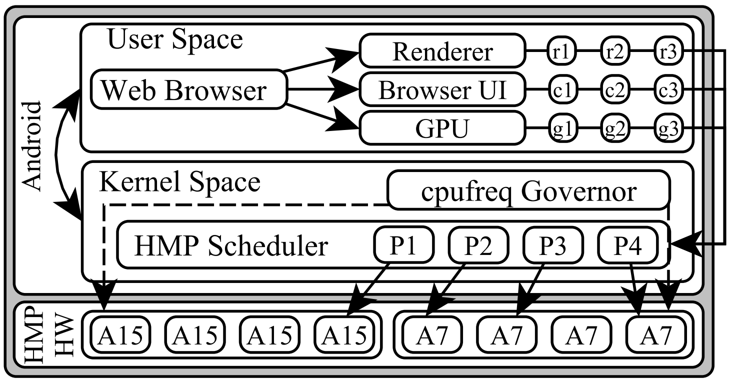


Figure 4.1: Web Browser with processes and threads running on an HMP platform.

rendering a web page depends on the type of the web page. But in most cases, the rendering engine and the JavaScript engine beneath it are the key components affecting performance and power consumption [154]. In order to meet the growing computational demand of mobile web pages, there has been a race by browser developers to enhance the processing speed. For example, Google's JavaScript engine V8 boosted JavaScript performance of Google Chrome by implementing a number of performance optimization techniques such Just–In-Time (JIT) compilation, inline caching, etc [42]. However, mobile web browsers are still designed assuming desktop conditions, that is, for performance, and little attention has been paid to power consumption for mobile scenarios. For example, Google's JavaScript engine V8 makes extensive use of performance optimization techniques such as JIT compilation, inline caching, concurrent garbage collection, and so on.

Hence, existing power management techniques for web browsing workloads on state-of-the-art Android systems leave much room for power optimization. Power management on Android systems is performed in collaboration between the Android governor which manages operating voltage and frequency, the scheduler, which allocates and schedules threads to each CPU core, and the power control unit which manages the power state of each CPU such as power down. However, the components are not designed in a way to minimize the power consumption, nor collaborate closely to reach a system-wide optimal solution. For example, the power control unit does not turn off unused CPU cores unless the whole device is left unused for some time, and the scheduler allocates and schedules tasks based on CPU usage thresholds not specifically taking into consideration the power consumption. Further, the Android default CPU governors are not aware of the performance requirements from the user so that they can conservatively reduce the operating frequency in order to optimize the response time. As a result, perhaps the most precious resource in a mobile system, the battery energy, is wasted in many real usage scenarios.

The poor interplay between power managing components becomes more distinctive when it comes to HMP platforms incorporating the big.LITTLE architecture as shown in Figure 4.1. This architecture is adopted in state-of-the-art smartphones like the Nexus 5X with its Qualcomm Snapdragon 808 processor [43, 131], the Samsung Galaxy S6 with its Exynos Octa 7420 [139, 138] and many more. The figure gives a complete overview of our evaluated system including the Exonys5422 SoC also based on ARM big.LITTLE architecture. It consists of two quad-core CPUs, of which one is a performance-oriented *big* CPU, and the other a power-saving *little* CPU. Individual CPU cores cannot be powered down due to complications in handing shared-cache, but the big CPU can be powered down as a whole. However, even if only one big core is on, all the other big cores have to idle which constitutes a significant portion of the total power consumption. Further, the thread allocation problem among the performance-oriented big cores, and power-saving little cores is not trivial. The default schedulers available in commercial products seek a rather simple solution based on setting a threshold value for CPU utilization. The HMP scheduler does not consider the full span of possible thread allocation and scheduling options such as consolidating workload on one CPU operating at high frequency and powering down the other cores as opposed to many cores running at low frequency.

In view of the previous discussion, to evaluate possible power saving potentials, this chapter provides a non-trivial, detailed analysis of the actual thread workloads generated by the web browser for a number of web pages. These new findings enable us to explore the potentials of power reduction on a real HMP platform. As we focus on the behavior of mobile web browsers, the web page rendering and JavaScript processing in particular, we use a trimmed-down version of the full Chrome browser, the Chrome *content shell*, which contains only the core components of the full browser and is referred to as *browser* in the following. The contributions of this work are as follows:

- We give a detailed analysis and characterization of the mobile web browser workload for loading a web page, by breaking down the browser CPU time and CPU energy based on the main browser processes and their threads for representative web pages. Here, *loading* refers to downloading, rendering and displaying the web page. We identify the process consuming most energy, which is the renderer (up to 70 %), and further break down its energy consumption by the website components HTML, CSS and JavaScript. Moreover, we find that different web pages have different workload distributions between the most relevant renderer threads.

- Based on the non-trivial analysis, we look into the potentials of power saving for mobile web browsing workload on HMP platforms using core allocation of individual threads, DVFS, and power gating. Considering the fact that we cannot power down individual cores of one CPU, we make the non-intuitive observation that not exploiting parallelism but consolidating all browser threads to one instead of all available big cores can lead to power savings with only small performance drop. Further, we show in a first attempt that we can save up to 39.21 % of the CPU power consumption when we power gate the big CPU after a web page has finished loading. Finally, we explore the DVFS power savings potential without performance loss during the web page loading using a power model. We find that we could save up to 26.6 % of energy consumption by applying more aggressive frequency down-scaling compared to the default power manager.

- We report a measurement infrastructure that we have developed for logging all the performance and power relevant information such as the core utilization, CPU frequency, power consumption, thread allocation, function tracing, etc, for the underlying HMP hard- and software platform. This infrastructure is a prerequisite for all analysis and characterization work as it enables us to find which CPU a thread is scheduled on - the most relevant information in case of HMP systems. It was used for all measurements presented in this chapter.

The rest of the chapter is organized as follows: Section 4.2 gives an overview of related work in the area of browser power management. In Section 4.3, we give details on browser internals and web browser application characteristics. We introduce our hardware and software measurement infrastructure in Section 4.4. In Section 4.5, we give a detailed analysis of the browser thread workloads and their energy consumption. Based on our results, we combine HMP platform specific power management with the workload characteristics of web browsers in Section 4.6.

4.2 Related Work

Although attention has mostly been paid to the performance of the mobile web browsers, researchers have recently begun paying attention to the power consumption of mobile web browsing. There are works putting emphasis on the network power consumption during web browsing. In [170], it was found that the coordination of the CPU's operating frequency and the network latency has significant impact on the energy consumption during web page loading as one has to idle wait for the other to complete its execution or transmission. Another work successfully reduced the power consumption by grouping the data transmissions during page loading and letting the 3G radio interface sleep more [168]. However, recent work has revealed that due to the significantly increased network speed, the complexity of the mobile web pages, and adoption of high-performance power hungry application processors to mobile platforms, the processor is becoming the major player in mobile web browsing both in terms of power and performance, and thus we focus on it. The measurements of mobile web page rendering power consumption showed that downloading and parsing CSS as well as JavaScript consumes a significant amount, up to 50 %, of total power [154].

There was some research characterizing the energy consumption of mobile web browsing according to web page primitives such as HTML, CSS and JavaScript. WebChar, a tool for analyzing browsers to discover properties of HTML and CSS that affect performance and power consumption, takes snapshots of a large number of websites and mines the model to produce a ranked list of expensive features in HTML and CSS [137]. It focuses on showing up energy pitfalls for web page design. In [154], the authors proposed power saving techniques based on web page modification and browser computation offloading to a remote proxy. However, they did not study browser workload characterization and power consumption on thread-level granularity, but a courser level of granularity, mostly according to web page primitives such as HTML, CSS and JavaScript.

There was an interesting line of work on this topic that studied web browsing power consumption on HMP platforms. In [171], a predictive model based on web page primitives was introduced. This model was used to find the appropriate core and operating frequency according to web pages in a heterogeneous system. However, they used a setup that consists of two separate platforms incorporating a big and a little CPU to validate their approach. This work does not reflect results for a real big.LITTLE platform because both CPUs are not on one chip. Another work identified QoS requirements of different mobile web applications by event-profiling to perform DVFS on a big.LITTLE platform [169]. Again, they did not observe actual thread-level workloads of web browsers. We took a cue from these studies and analyzed the different processes and their threads with the JavaScript engine to evaluate their power consumption.

In summary, the main contribution of this chapter is a detailed, non-trivial characterization of the web browser workload and energy consumption at thread-level granularity, whereas previous work operates at a more course-grained application level. The workload analysis of the browser at that level of granularity cannot be translated directly into power management policies as we are able to do in this work, e.g., determining power-aware thread allocation schemes to CPU cores. Moreover, to the best of our knowledge, we are the first to show power saving potentials and propose power management techniques for web browsers on an HMP platform by exploiting all available mechanisms on the platform, such as power gating, DVFS and HMP scheduling.

4.3 Page Rendering in Web Browsers

This section gives an overview of the elements of a web page, the structure of a web browser and the role of the JavaScript engine. Further, we discuss the workload characteristics that are specific for web browsers and need to be considered for power management.

4.3.1 Components of a Web Page

A web page consists of static and dynamic elements. Static elements are described by HTML and CSS. HTML describes the basic structure of a web page whereas CSS defines its styling. Scripting languages like PHP and JavaScript are used for dynamic and interactive elements such as user inputs or slide shows. We focus on JavaScript as research has already shown its large impact on web browser power consumption [154]. JavaScript intensive phases occur during the loading of a web page or user interaction.

4.3.2 Components of a Browser

The main components of a browser are the browser engine, the rendering engine and the JavaScript engine as shown in Figure 4.2 [40]. The browser engine acts as an interface between user inputs and the rendering engine. When a web page is parsed, the rendering engine creates a so-called Document Object Model (DOM) tree from the HTML code. It also parses the CSS into style rules. DOM tree and style rules are combined to the *render tree*, the internal representation of a web page. Hereafter, the exact positions of the render tree components are determined. Finally, the web page can be drawn on the screen. JavaScript code is processed by the JavaScript engine and manipulates nodes of the DOM tree.

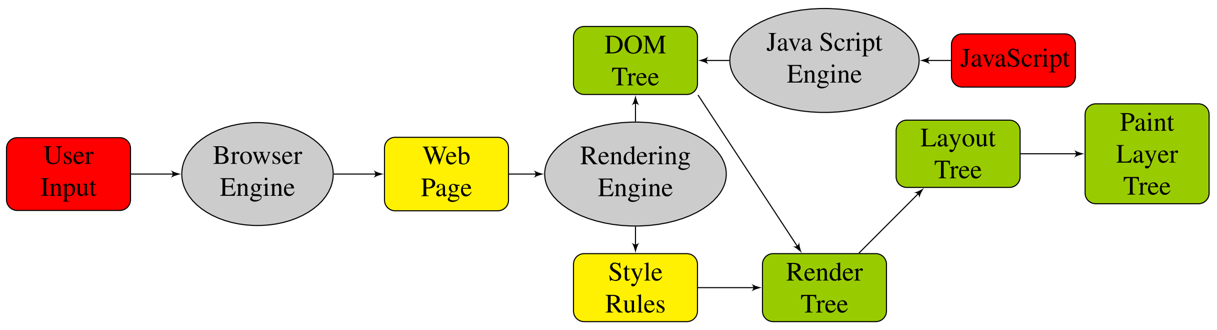


Figure 4.2: Schematic structure of a browser.

4.3.3 JavaScript Engine V8

As our experiments were performed using the Chrome browser, we focus on describing the internals of V8 [42], the JavaScipt engine used in this browser. The browser itself executes three main processes of which one corresponds to the browser engine, one to the rendering engine and one to the painting task that communicates with the GPU. V8 executes as part of the renderer. Our measurements have shown that the rendering process consumes up to 70% of the total CPU time, depending on the web page, where up to 60 % of the rendering energy is due to the JavaScript engine.

JavaScript is an untyped script language. The code needs to be downloaded, parsed, compiled and executed. V8 can perform all these stages partly concurrently. At the time of performing the experiments in this chapter, it is a compile-only JavaScript virtual machine consisting of a quick, one-pass (baseline) compiler and a more aggressive optimizing compiler. The baseline compiler performs compilation on the main thread whereas optimized code is compiled by concurrent compilation threads. Finally, V8 incorporates a multi-generational garbage collection mechanism that can be triggered in parallel to the main thread execution.

4.3.4 Web Page Rendering

The browsing process can be separated into a loading phase and a post loading phase as depicted in Figure 1.4. The loading phase is defined as the phase before the *loadEventEnd* function call of the main frame occurs. During the loading phase, the website needs to be downloaded, rendered and displayed. These steps are highly resource intensive. For this phase, JavaScript plays an important role as it is used in most of the popular websites and consumes a large amount of energy [154]. The goal during the loading phase is to download, render and display the page as fast as possible, spending as little energy as possible. During the post loading phase, the resource requirements vary from website to website. Among the 25 most popular websites based on rankings from Alexa Internet [4], which provides commercial web traffic analytics, are search engines, social networks, online shops, and encyclopedias such as Wikipedia. The workload highly depends on the type of web page and its degree of interaction with the user, e.g., scrolling. Other aspects can be the amount of JavaScript executed in the background, animations or video streaming.

The corresponding browsing phase and website characteristics could be exploited to design a web browser specific CPU power management unit that performs in an optimized way compared to the standard Android power manager.

4.4 Measurement Setup

We have implemented a software measurement framework on top of the commercially available Odroid-XU3 hardware platform, which we have already described in Section 1.1.2. The framework is capable of capturing the power consumption of CPU clusters with a granularity of 1 kHz, and the CPU usage of individual threads running on each core with a granularity of 20 Hz. With this setup, we can not only study the overall power consumption, but also the detailed internal traces of web browser threads.

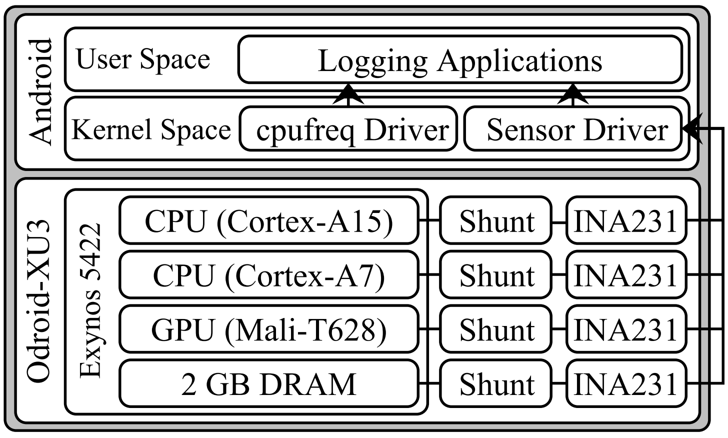


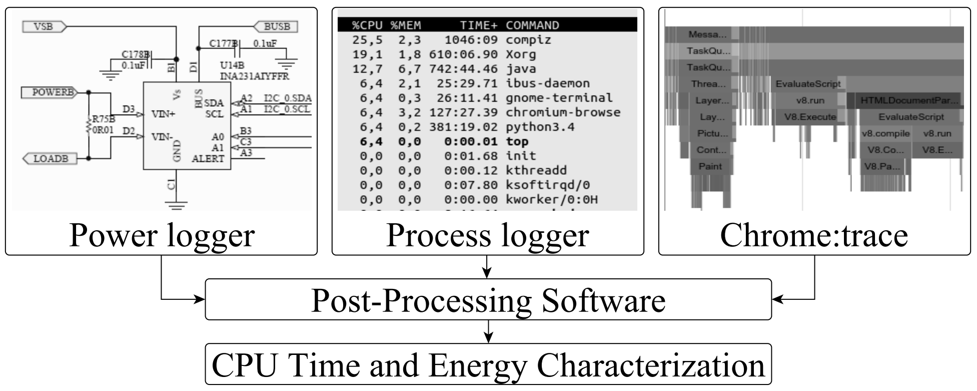
Figure 4.3: Exynos5422-based measurement setup.

4.4.1 Hardware Infrastructure

The Odroid-XU3 board provides a built-in power measurement interface which has been utilized in our experimental setup as shown in Figure 4.3. Shunt resistors are placed in front of both CPUs, the GPU and the memory. INA231 sensors measure voltage and current at the shunts of the target component while the kernel driver calculates the power. Further, the details of the Exynos SoC of the Odroid-XU3 board have already been described throughly in Section 1.1.2 and Section 3.5.2. Please refer to those sections for details on the development board.

4.4.2 Software Infrastructure

Our software setup is a combination of three different logging environments as shown in Figure 4.4, (1) the power logger, (2) the process logger and (3) the Chrome:trace environment. Combining all the information, we are able to get a detailed profile of which thread was running when, on which core and how much energy it consumed at that point in time. We exploit this



```
%CPU %MEM     TIME+ COMMAND
25,5  2,3   1046:09 compiz
19,1  1,8 610:06.90 Xorg
12,7  6,7 742:44.46 java
 6,4  2,1  25:29.71 ibus-daemon
 6,4  0,3  26:11.41 gnome-terminal
 6,4  3,2 127:27.39 chromium-browse
 6,4  0,2 381:19.02 python3.4
 6,4  0,0   0:00.01 top
 0,0  0,0   0:01.68 init
 0,0  0,0   0:00.12 kthreadd
 0,0  0,0   0:07.80 ksoftirqd/0
 0,0  0,0   0:00.00 kworker/0:0H
```

Figure 4.4: Software setup for data acquirement.

data to identify the effects of DVFS and thread allocation on the performance of the browser. The Chrome version that we run is 61.0.3139.0.

Power Logger

For the power measurement, we have developed a logger which instruments the underlying kernel driver of the sensors. It acquires the power of both the A7 and the A15, GPU and RAM. Besides the power, it enables us to measure A7 and A15 CPU utilization for each of the individual cores at a sampling frequency of approximately 1 kHz. Further, we log the frequency of each CPU by instrumenting the *cpufreq* driver.

Process Logger

Moreover, we have developed a process tracer for capturing information about the individual processes and sub-processes of applications. The logger is a C-program capturing the accumulated CPU time and the core a process is currently scheduled on. This is necessary to identify which threads are running on the A15 and which on the A7. Without this information it is not possible to create a power profile on an HMP platform. While the CPU allocation is not relevant to extract power information per thread for traditional chips with only one CPU, it is crucial for the big.LITTLE architecture.

Chrome:trace

To get deeper insight into the logged processes during the execution of the browser, we instrumented the Chrome:trace framework. It gives detailed stack traces of which functions were executed when and which process they belong to. In this way, we can identify the threads that are executing HTML, CSS or JavaScript. Chrome:trace does not give any information on the thread core allocation.

4.5 Web Browser Workload Characterization

In this section, we present a detailed analysis of the web browser workload for the loading and post loading phase on the underlying HMP platform. We look into the CPU time and energy consumption of the browser threads to identify power saving potentials by thread-to-core allocation, DVFS and power gating. To the best of our knowledge, this is the first work to analyze the actual thread workloads generated by web browsers for power management. Further, we look more closely on rendering and JavaScript-related power consumption. The representative websites we have chosen based on rankings from Alexa Internet are eBay, Amazon, Reddit, Facebook, Wikipedia and CNN. We initiate the loading of the web page and wait for 10 seconds in each experiment.

4.5.1 Breakdown Analysis of Browser Threads

As discussed before, the chosen browser consists of three main processes: (1) the browser process itself, which handles user inputs, (2) the rendering process, which sets up the frames, and (3) the GPU process, which triggers the GPU to draw frames on the screen. All of them create a set of threads, of which the two most important ones in terms of the workload they generate are the main renderer thread, *CrRendererMain*, and the compositor tile worker, *CompositorTileW*. Both belong to the renderer process. The main renderer thread sets up the web page including HTML, CSS and JavaScript while the compositor tile worker deals with GPU communication. It is of major importance to identify the critical threads related to energy consumption. This knowledge enables us to apply advanced power management strategies such as power-aware thread to core allocation.

Figure 4.5 on the top shows the time distribution of the three main browser processes for A15 and A7, respectively, loading the representative web pages with the default Android settings. The bars are split up among the three processes of the browser where the renderer process is further split up into the main renderer thread, the compositor tile worker and other miscellaneous threads. The relative CPU time is normalized to the total A15 CPU time on a per web page basis for visualization purposes, as the absolute values of the CPU time, e.g., for CNN and eBay, are significantly different. The main observation is that the renderer process, which mainly consists of the CrRendererMain and the CompositorTileW threads, takes up most of the A15 time, and hence, contributes the most to the energy consumption as depicted in Figure 4.5. This is the first work to perform per-thread analysis of a mobile web browsing workload, which explicitly shows different levels of parallelism among different threads. This important information enables us to target the major power consuming browser threads for energy reduction.

Furthermore, we observe from Figure 4.5 (top) that different web pages exhibit different degrees of thread-level parallelism. For example, the time spent on the main renderer thread and the compositor tile worker thread is almost the same for eBay. In case of Amazon and Facebook, compositor tile worker thread time is only 20-25 % of the main renderer thread time. In case of CNN and Wikipedia, the main renderer thread dominates the execution time. The power saving technique should be aware of the thread-level parallelism and perform core allocation and workload consolidation accordingly, as we are considering an HMP platform comprising multiple CPUs. In Section 4.6.2, we are able to show how the number of schedulable cores affects the page loading time and energy consumption according to web pages exhibiting different degree of thread-level parallelism based on our observations.

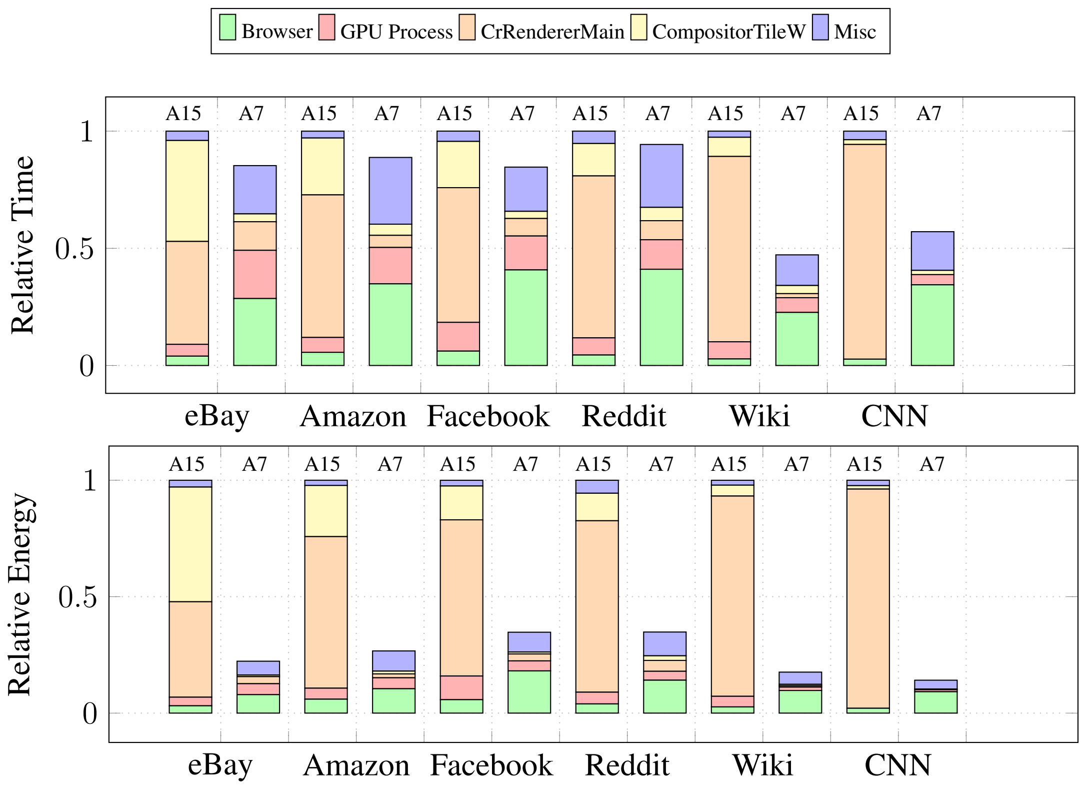


Figure 4.5: Relative CPU time (top) and CPU energy (bottom) consumption web browser threads per A15 and A7 cluster for representative websites.

Also, we observe that the CPU time spent on the A15 is only between 50-70 %, whereas it contributes between 80-90 % towards the total energy consumption. This is expected because the A15 is designed in a performance-oriented way. Figure 4.5 (bottom) shows that the A15 is approximately 3 times more power consuming than the A7. Therefore, in order to save power, it is preferable to only allocate threads on the A15 that are the bottleneck for achieving the performance requirement. We also investigate the impact of deferring thread execution on A15 on power consumption and web page loading time in Section 4.6.2.

4.5.2 Rendering Process

As shown in the previous section, the rendering process is the most time and energy consuming process. In this section, we further analyze the energy and time contribution of different web page components handled by the renderer, especially focusing on JavaScript since it contributes most to the rendering energy consumption. As mentioned before, for the Chrome browser we are experimenting with, JavaScript code is handled by the JavaScript Engine V8. Therefore, we refer to all JavaScript related calls as *V8* in the following. Figure 4.6 depicts the energy consumption of the rendering process divided by the main web page components CSS, HTML and V8 described in Section 4.3.1. The figure shows that V8 consumes a significant part of energy, depending on the website. For eBay, V8 takes up about 25 % of the total rendering energy while it takes up to 60 % for CNN.

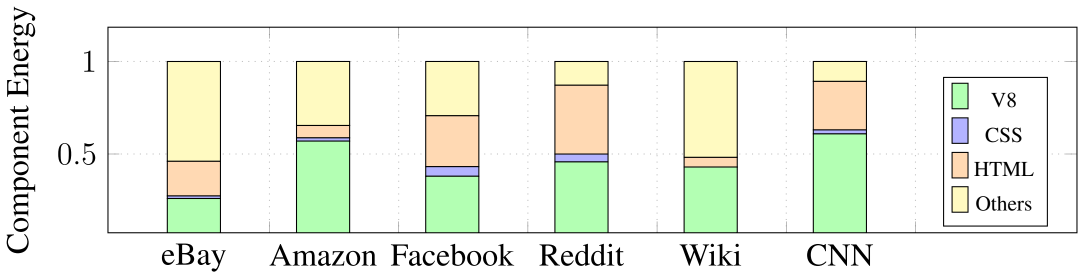


Figure 4.6: Relative energy distribution of the web page components HTML, CSS and JavaScript.

V8 function and thread time analysis

We have found that V8-related functions consume a significant amount of energy during web page rendering. To identify bottlenecks and power optimization potentials, we have investigated the time distribution of different V8 execution stages as described in Section 4.3.3 and studied the distribution of V8 workloads over threads within the rendering process.

Figure 4.7 shows the relative time V8 spends in its working stages parsing, compilation, execution and garbage collection. For most of the pages, the time distribution is very similar. We see that V8 spends up to 40 % in parsing and compilation while it spends 50-60 % in the execution stage. The stages alternately occur on a time-scale of micro- to milliseconds. The results show that a large amount of time and, consequently, energy is spent on preparing the JavaScript code for execution rather than actually executing it. In other words, the reason why parsing and compilation takes that much time should be investigated further. Our results emphasize the importance of designing the JavaScript engine in a power-aware fashion. For example, information which can be gathered about the execution at parsing and compilation stage could be later exploited for power management.

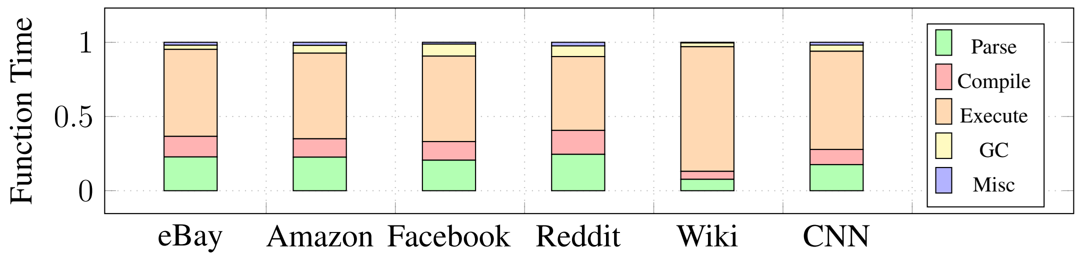


Figure 4.7: Relative time distribution of V8-related function calls by category.

Moreover, we have looked into separate threads that are executing V8-related work and their distribution across the CPU cores. We have found that between 83-96 % of V8 is executed within the main renderer thread for the representative websites. Further, note that 1-13 % of the thread time is used by a *ScriptStreamerThread* which parses JavaScript code. This is important as this is the only other V8-related thread running on the A15 besides the main renderer thread, hence, one of the most energy consuming V8 threads. It takes up to 5 % of the total A15 energy consumed by V8. Other threads that are, e.g., responsible for recompilation of the JavaScript code use at maximum 4 % of the execution time. This information can be exploited for power saving by thread allocation, for example moving the ScriptStreamerThread to the A7 considering its penalty on performance.

4.6 Power Management for Web Browsers

The default Android power management, which is designed for a wide range of applications, leaves much room for further power reduction in case of the web browsing workload in specific. First, the HMP scheduler distributes threads over as many CPU cores as possible to exploit parallelism whereas the power-optimized thread-to-core allocation depends on the performance requirements of a web page. Second, the most popular Android default governors, such as *interactive* and *ondemand* governors, are biased towards the performance requirements of the web browser, and, hence, the operating frequency is reduced too conservatively. Third, the default power management policy is not tuned for HMP platforms such that it does not consider power gating while an application is running. In the following, we apply different power management strategies for the mobile Chrome browser. Based on our characterization results in Section 4.5, we show a non-intuitive power-aware thread-to-core allocation strategy in Section 4.6.2 and outline the potential for energy savings in Sections 4.6.3 and 4.6.4.

4.6.1 Power-Performance Trade-off Analysis

In this section, we investigate the trade-off relationship between the power consumption and the loading time of the representative websites. We limit the maximum CPU frequency of the A15 to various values and let the default governor control the operating frequency below that value. As explained in Section 4.3, the loading time of a web page is defined as the time duration from the start of loading the page until the *loadEventEnd* function is called.

Figure 4.8 shows the loading time and energy consumption for the test scenario described in Section 4.5 where all A15 and A7 cores are schedulable and the maximum operating frequency of the A15 CPU is capped to the corresponding values on the x axis. In case of eBay, by capping the maximum frequency to 1.2 GHz, the total energy consumption is reduced by 34.6 % while the loading time is increased only by 16.7 % (0.6 s), which is marginally perceivable by the user. In case of Wikipedia web page loading, sacrificing only 1 s of loading time saves over 30 % of energy.

4.6.2 Constraints for Core Allocation

In this section, we look into the effect of thread allocation to cores. Therefore, we investigate three different cases: Running the browser threads on all A15 and A7 cores (case 1), on one A15 core and all four A7 cores (case 2), and the four A7 cores only (case 3). Controlling the number of schedulable cores is done by setting the the processor affinity of the threads such that the

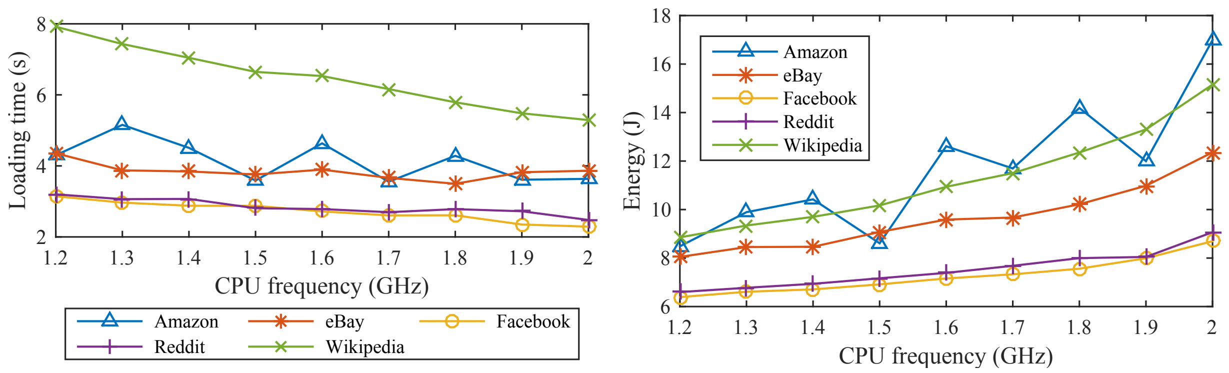


Figure 4.8: Web page loading times (left) and A15 energy consumption (right) according to different A15 frequency cap.

HMP scheduler allocates the threads solely to the desired cores. In this analysis, we select two web pages, eBay and Wikipedia, that exhibit different characteristics in terms of thread-level parallelism. As can be seen in Figure 4.5, eBay exhibits even CPU time distribution among the two threads CrRendererMain and CompositorTileW, while only the CrRendererMain dominates the CPU time for Wikipedia. We observe that consolidating the workload into a smaller number of cores is more efficient in terms of energy consumption than distributing the workload over multiple cores.

Comparison of case 1 and case 2 for eBay

Figure 4.9 shows the eBay loading phase for case 1 (top) and for case 2 (bottom). Obviously, the A15 CPU utilization goes up to 200 % for case 1, while the value is limited to 100 % for case 2. The power consumption of the A15 is coupled to the CPU utilization changes. The A15 power consumption goes up to 5 W for case 1 while it is clamped around 2 W for case 2. This computes to the significant difference in the total energy consumption, which is 13.31 J for case 1, but only 11.57 J for case 2, 13.1 % less. However, the increase in loading time is marginal from 3.6 s to 3.8 s (5.6 %), which is not significantly perceivable by the users. Besides the effect on the energy consumption for case 2, it is also important to investigate the effect of smoothing the power curve on the overall battery lifetime. It is well known that high peak current flows have a negative impact on the battery lifetime. We leave such an analysis as a future work.

Comparison of case 1 and case 2 for Wikipedia

The power consumption and CPU utilization while loading the Wikipedia web page is shown in Figure 4.10. In contrast to eBay, less degree of thread parallelism exists in the Wikipedia rendering workload, and, hence, the usage stays consistent around 100 % for both test cases. This fact is also reflected in the power graph such that the power consumption remains around 2 W for both cases, which differs significantly from the eBay web page rendering. The energy consumption and the loading times are also very similar, 15.40 J and 5.4 s for case 1 and 15.26 J and 5.4 s for case 2. The comparison between case 1 and case 2 for eBay and Wikipedia web page rendering shows that controlling the number of utilized A15 cores has different impacts on

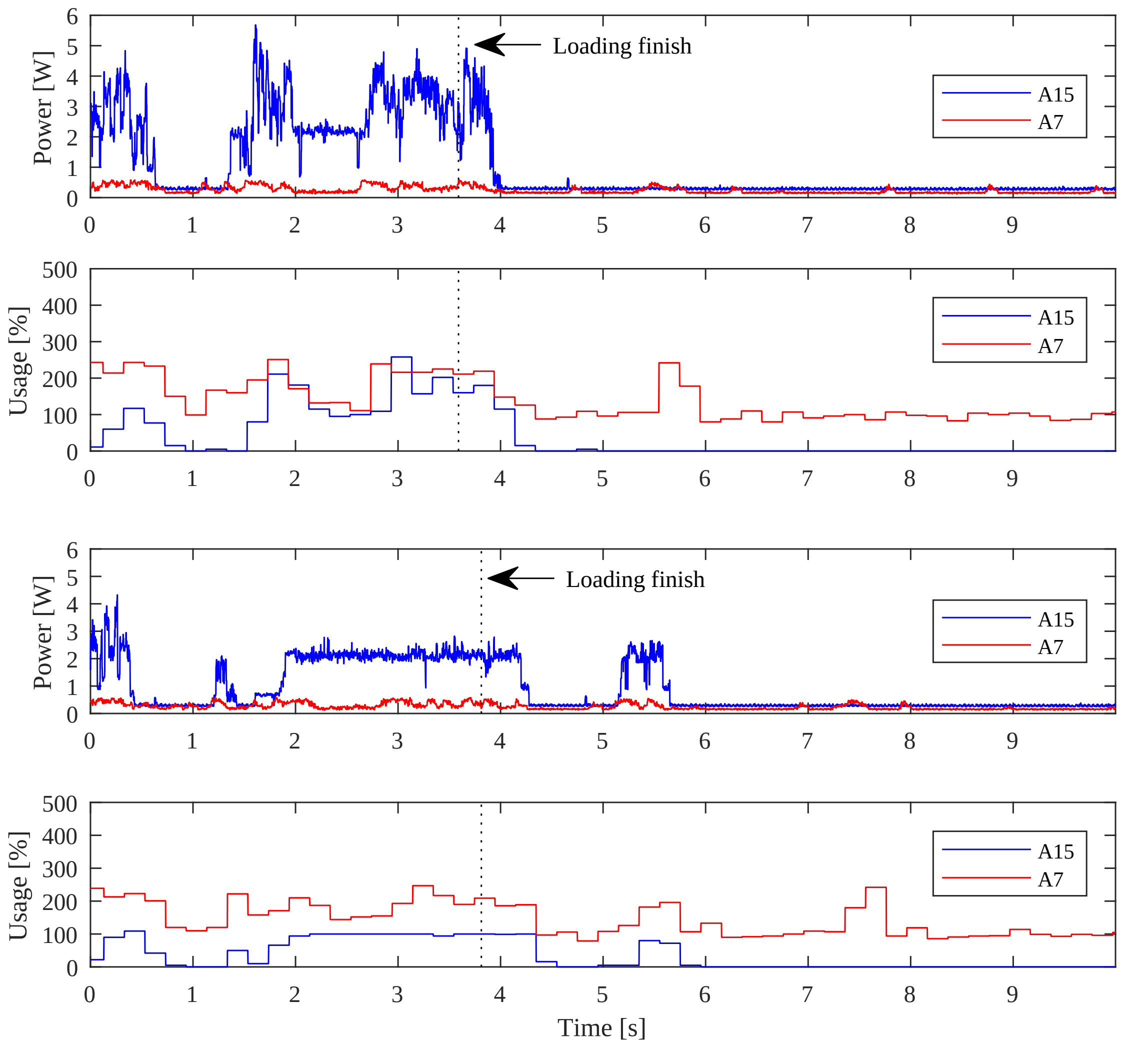


Figure 4.9: Power consumption and CPU utilization for loading eBay for case 1 (top, energy: 13.31 J) and case 2 (bottom, energy: 11.57 J).

power consumption depending on the degree of thread-level parallelism. Nevertheless, using less number of performance oriented A15 cores is generally preferred even if there is sufficient thread-level parallelism because of potential savings in the energy consumption (13.1 %) compared to a marginal increase in loading time (0.2 s, 5.6 %). This is a notable and non-intuitive observation as it is natural to expect significant performance improvement if more cores are utilized.

Case 3 for eBay

In this case, we power gate the complete A15 and use only the four A7 cores to load the web pages. The A15 utilization is zero all the time because all threads run on the A7 as shown for eBay in Figure 4.11. The power consumption of the A7 is nearly the double of the cases 1 and 2, but the total power consumption value of case 3 is significantly smaller compared to the

A15 power consumption in the above cases. We observe that the A7 consumes around 0.5 W during the loading phase and 0.3 W during the post loading phase.

The overall results are summarized in Table 4.1. As we have described in the above analysis, comparing case 1 and case 2, the loading times increase marginally if less number of A15 cores are utilized, but there could be more reduction in energy consumption depending on the thread-level parallelism of web pages. As for case 3, the loading time roughly increases by a factor of 2 compared to the cases 1 and 2, but even more energy could be saved by using A7 only. If we make a careful evaluation of the user requirement during web page loading, core allocation could be used to leverage power consumption at cost of a marginal loading time increase.

	eBay	Amazon	Facebook	Reddit	Wikipedia	CNN
Case 1	3.6 s 13.31 J	3.6 s 16.99 J	2.3 s 8.70 J	2.6 s 9.07 J	5.4 s 15.40 J	13.0 s 25.35 J
Case 2	3.8 s 11.57 J	3.6 s 15.95 J	2.5 s 8.37 J	2.7 s 9.08 J	5.4 s 15.26 J	13.9 s 24.07 J
Case 3	6.8 s 4.87 J	5.2 s 5.37 J	5.1 s 4.39 J	5.1 s 4.42 J	11.5 s 5.20 J	> 13 s

Table 4.1: Loading times and energy consumption of representative websites for different core configurations.

4.6.3 Power Savings by DVFS without Performance Compromise

As discussed in Section 1.1.3, default governors for Android often fail to assign energy-optimal frequencies to the CPUs. However, prediction of the exact workload and setting the optimal frequency for a web browsing workload are difficult tasks to achieve. In this section, we make a rough estimate of how much potential exists for power savings without performance loss by applying DVFS. Figure 4.12 shows the estimates of power consumption, CPU utilization, and operating frequency if an *oracle* workload predictor was used. The oracle predictor is a theoretical construct of which we assume is capable of knowing the exact future workload such that the utilization of the core that executes the bottleneck thread is kept as close as possible to 100 % by applying DVFS. In other words, it finds the lowest possible CPU frequency that does not result in a performance loss unlike the performance oriented default Android governors. The power graph in Figure 4.12 is obtained by using the following CPU power model

$$P_{cpu} = u \cdot C_{eff} \cdot V^2 f + P_{static}(V), \tag{4.1}$$

where u is the sum of utilization of the cores, C_{eff} is the effective switching capacitance, and V and f are the operating voltage and frequency, respectively. We fit the model to the measured power consumption of the A15 processor and find that 1.0158×10^{-9} F is a reasonable value for C_{eff}. The analysis shows that if we were to predict the workload precisely, the total energy consumption could be reduced from 10.889 J to 7.996 J, which is about 26.6 %. We consistently observe that Android governors are conservatively tuned such that they do not respond fast enough to web browser workload variations.

4.6.4 Post-Loading Phase Power Gating

We consistently observe that in most cases the A15 is not being utilized and the A7 is mostly handling the rendering workload during the post-loading phase. However, the Android default power managers never apply power gating to the A15 as long as the device itself is in use. This leaves scope for power gating techniques to be utilized during the post-loading phase. The power consumption of the A15 during idling is approximately $P_{idle} = 0.27\,\mathrm{W}$, while it is only $P_{off} = 0.04\,\mathrm{W}$ when power gated. Hence, power gating results in $85\,\%$ idle power savings. Although the absolute idle power is almost negligible compared to the active power, the energy consumption of the A15 during the post-loading phase could be significant depending on the user activity, e.g., the user may read an article for a considerable amount of time after the web page loading finishes.

We implemented a simple prototype power manager that power gates the A15 immediately when the utilization is zero during the post-loading phase and turns the A15 back on when the A7 utilization rises above $110\,\%$. The threshold of $110\,\%$ is set because we observe that the A7 workload during the post-loading phase was fairly single-threaded, so that turning on the A15 cores would benefit in terms of performance. A real world power manager featuring power gating should allow for well established theory on predicting idle time and breakeven time as well as practical constraints such as granularity of power gating, in our case the CPU clusters. The prototype power manager is very naive, and, hence, cannot be applied for browser power management in general, but suffices for two simple usage scenarios. We repeat the experiments as described in Section 4.5 for Wikipedia and eBay using the prototype power manager. For Wikipedia, the loading time takes 5.4 s and consumes 13.84 J. There is no increase in loading time, but the energy consumption decreased by $10.1\,\%$ compared to the case without power gating (15.40 J). For eBay, we observe a loading time of 4.8 s and an energy consumption of 8.09 J. Although the increase in loading time compared to the default settings is 1.2 s (25 %), we can achieve energy saving of 39.2 %. These results show a large scope for power savings by utilizing A15 power gating for web browser workloads. However, a more elaborate power gating technique requires detailed knowledge of the time and power overhead, which we leave as a future work.

4.7 Consequences for Browser Power Management

In Section 1.1.4, we have shown that the default Android power management does not perform well as the three power managing entities, the governor, scheduler, and power control unit separately manage the operating frequency, thread allocation/schedule, and power state of the CPU, respectively. There has been a significant amount of theoretical research on co-optimizing thread allocation and DVFS on HMP platforms, and the development of the so called *Energy Aware Scheduler* [82] for big.LITTLE systems, an ongoing project pushed by ARM and Linaro. However, an integrated power manager capable of actually performing all the policies has not yet been implemented, especially in the domain of mobile web browsing. In the future, we would need a power manager that either integrates the separate components or lets them closely collaborate together to minimize the power consumption. This power manager should be aware of the different phases of web page rendering and the user requirements to perform optimal power management. Moreover, it should be aware of different computation demands and impacts on user experience among threads. It would enable us to selectively allocate performance critical threads such as the *CrRendererMain*, to the appropriate cores. However, threads that produce a high workload but are not critical for fast web page rendering could be deferred to power-optimized cores. A part of our future work will be to identify performance critical threads and perform scheduling and DVFS to maintain a good user experience.

4.8 Summary

This chapter provided a detailed look into web browser workloads on HMP platforms and seeked potential power savings based on the observations. Unlike previous works that analyzed the power consumption according to the inputs to the web browser, the new aspect of this work is the focus on the actual thread workloads and function calls invoked by the web browser. They provide information that can be used directly for power management. We performed a breakdown analysis of the CPU time and power consumption per CPU of the browser processes and threads among websites. Furthermore, we performed a case study on how the operating frequency, number and types of schedulable cores affect the power consumption and performance of a mobile web page rendering. Based on the characterization, we applied several power management techniques, such as DVFS, thread allocation to CPU cores and power gating. Moreover, we outlined the theoretical power saving potential for web browsing in Android. We showed that there is significant scope for power management compared to the Android default governors. Our initial results show that current Android power management leaves a significant room for improvement and relevant operating system entities, the governor, scheduler, and power control unit, should work collaboratively to achieve higher power savings. We use these findings to implement our own browser-aware governor that we will describe in the next chapter.

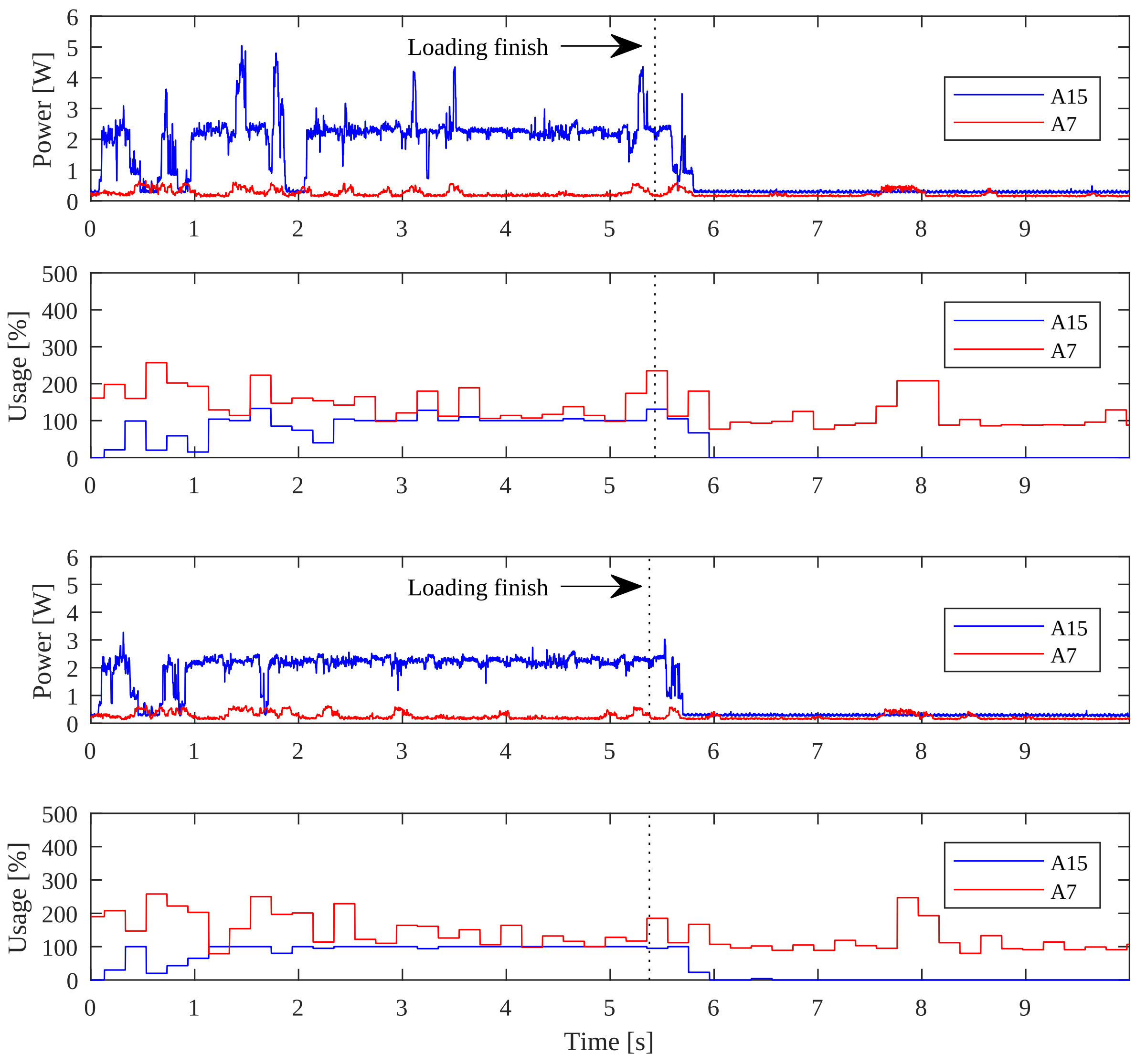


Figure 4.10: Power consumption and CPU utilization for loading Wikipedia for case 1 (top, energy: 15.40 J) and case 2 (bottom, energy: 15.26 J).

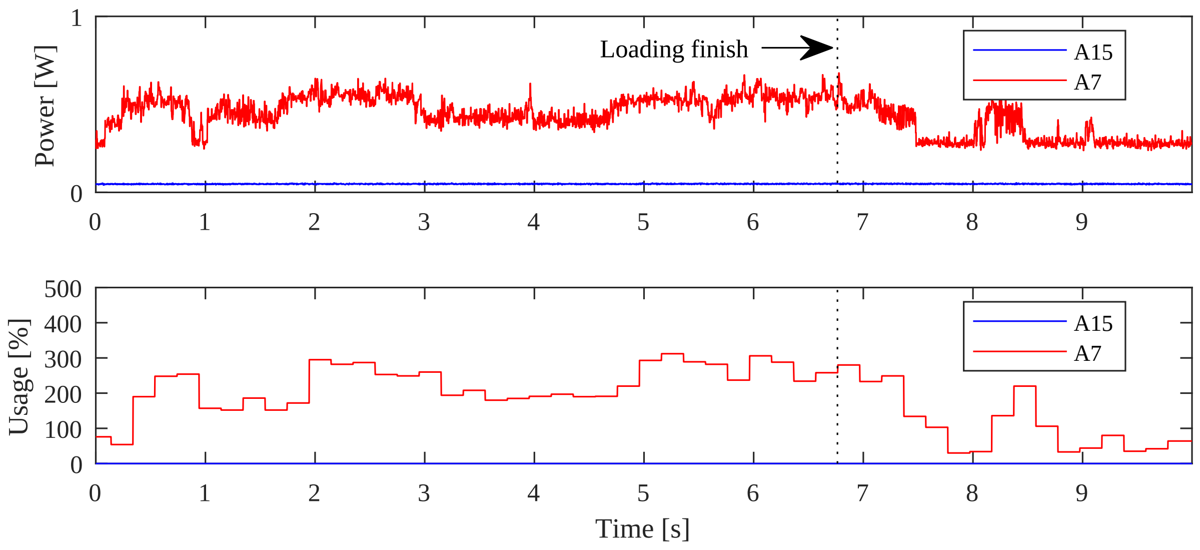


Figure 4.11: Power consumption and CPU utilization for loading eBay for case 3.

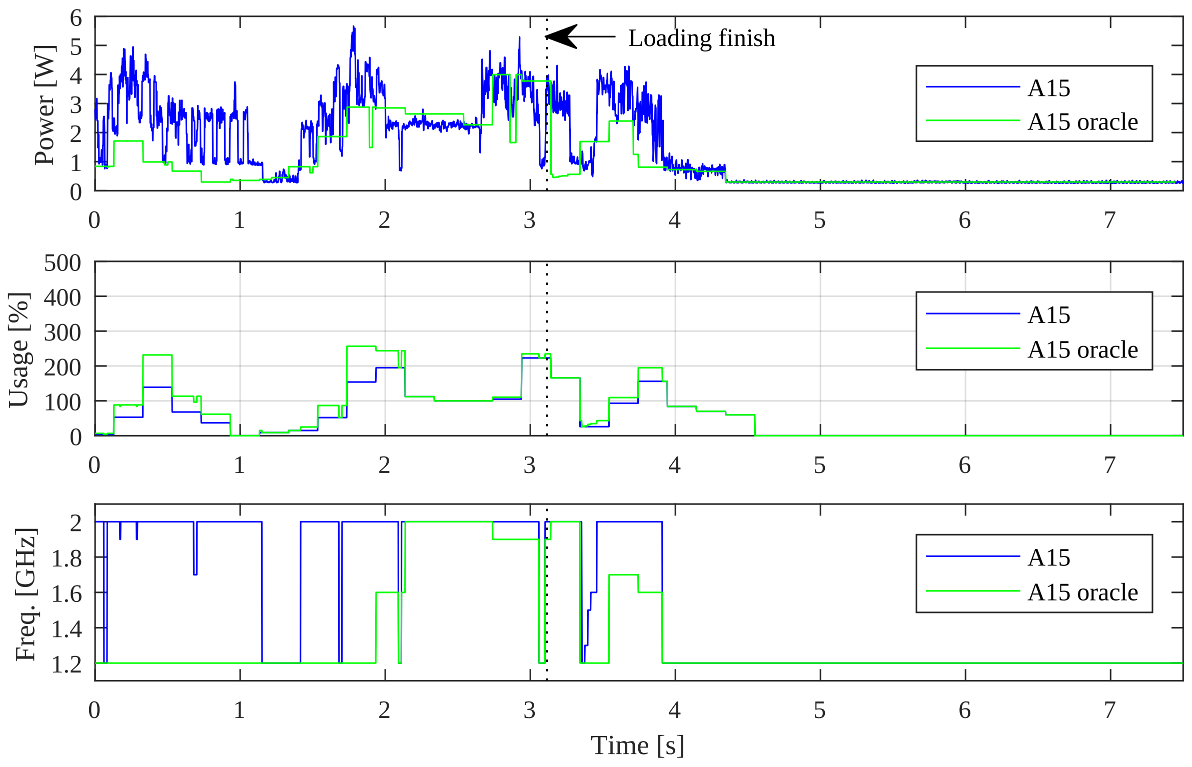


Figure 4.12: Frequency, usage and power of the A15 estimated by the oracle verses a real measurement when loading eBay.

5 Phase-Aware Web Browser Power Management

Based on the findings in the previous chapter, we implement and evaluate our own browser-specific and browser-aware governor in this chapter. This part of the work helps us to gain insight into the characteristics of different states of Android applications. We study different states such as the scrolling or the loading state and their performance and power requirements. Based on the performance requirements, there are different power management strategies that can be applied to the various states. The states are not specific to the web browser, but also re-appear in other applications, e.g., scrolling is also available in social media application and loading can be found in any rendering intensive application. Consequently, the findings in this chapter are not only meaningful for web browsers but for Android apps in general. They enable us to generalize the approach and create power management strategies that are suitable for more than one application.

5.1 Introduction

The time we spend on mobile devices has recently surpassed the time spent on desktop computers [98]. However, using mobile devices for daily activities like instant messaging, social networking, and web browsing always involves being connected to the Internet. Consequently, the smartphone constantly satisfies our need for instant messaging, shopping and information. 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 [44]. 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

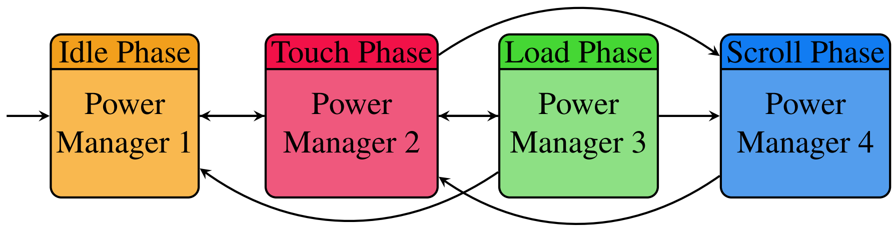


Figure 5.1: Proposed phase-aware power management.

pages [47, 164]. Hence, modern web browsers usually target a frame rate of 60 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. The web browser existed long before modern smartphones and tablets were available on the market. It emerged with the rise of the Internet. 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 [40]. Such complexity, as well as the focus on performance, are because many mobile browsers have evolved from the desktop world.

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 [127]. While power management of mobile devices has been extensively studied for several years now, the focus has been on video decoding applications [66, 134], and also on games [35, 79]. Surprisingly, power management techniques specifically targeting the web browser have been less studied [172, 169, 124]. But both, its importance, and its potential for power savings, is being increasingly recognized.

Android Power Management on HMP platforms

While a mobile device is comprised of 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 [18]. 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. For example, if the screen is very bright, it will consume more power and 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 OS. They respectively determine at which voltage and frequency the CPUs should operate, which threads run on which CPU cores and which CPU cores are active. The most popular governors in Android are the *ondemand* [113] and the *interactive* [15] governors. They perform DVFS according to the CPU's utilization, with a focus on the responsiveness to user inputs by ramping up the frequency quickly and conservatively reducing it. Moreover, they do not power down (*power gate*) CPU cores during runtime as the penalty for turning the cores back on affects the system's 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 between them. In particular, this results in energy wastage on HMP SoCs with big.LITTLE architectures, which are widely used in modern smartphones, such as the Nexus 5X (Snapdragon 808) [43, 131], the Samsung Galaxy S8 (Exynos 8895) [141] and even the iPad pro [159].

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's 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 when it is unnecessary to finish them early. Such actions can significantly increase the power consumption and reduce a 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 [30, 91, 34, 35].

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 workloads and performance requirements [35]. During the loading phase, frames need not to be updated as frequently as during the playing phase as 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 6.2, is proposed as a basis for power management in this work.

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 [59] (see Section 1.1 for details). The board features a power-saving CPU (A7) and a performance-oriented CPU (A15). Figure 5.2 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 top show the results for the ondemand governor, while the ones on the bottom 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 5.5 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's 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 chapter – 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.

The main contributions of this chapter 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 work, is acceptable in the case of mobile devices, since unlike in desktops

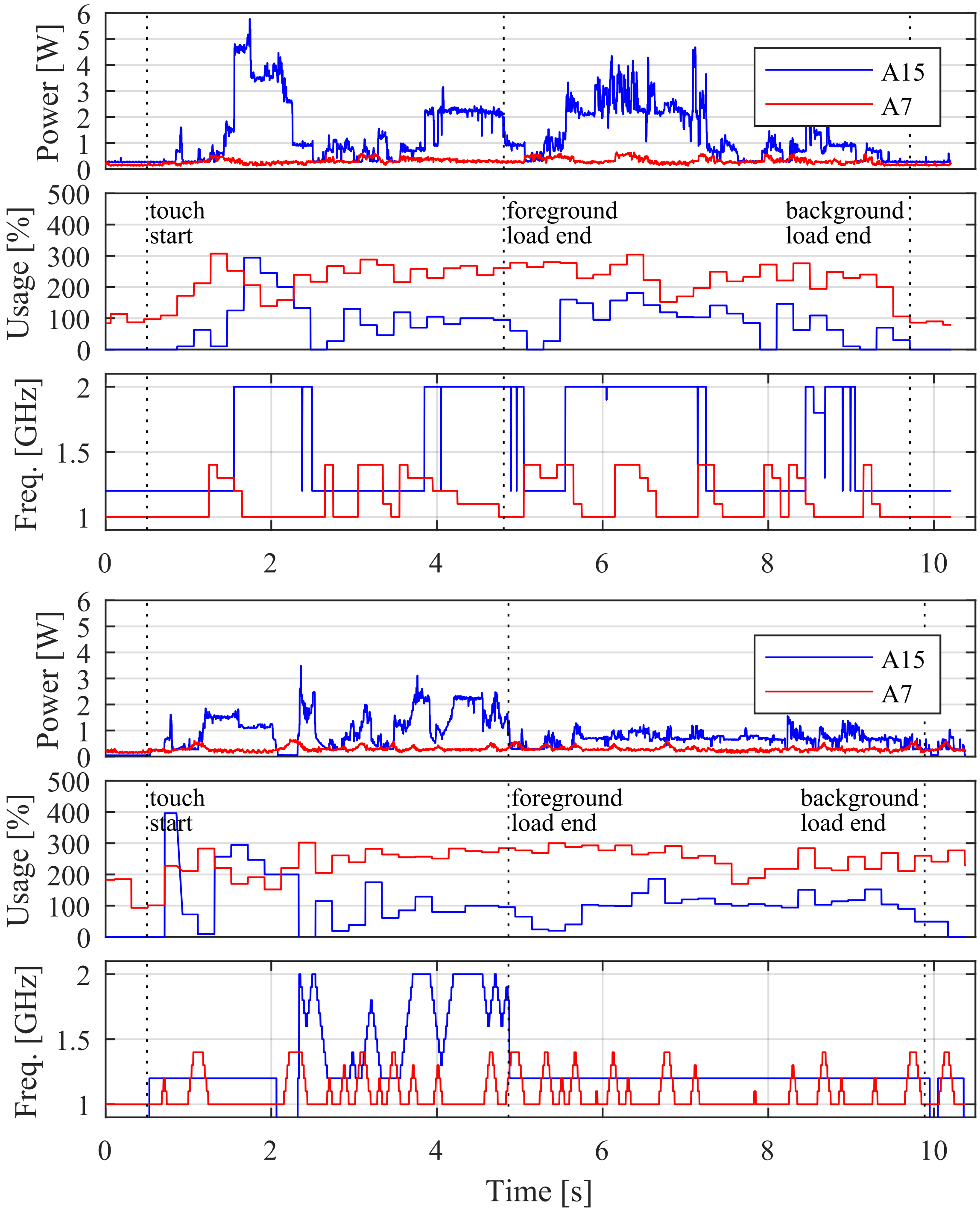


Figure 5.2: Power, CPU usage and CPU frequency for loading the Reddit page with ondemand (top) and browser governor (bottom).

or laptops, they usually run only 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 work might become relevant in future mobile devices.

The rest of the chapter is organized as follows. Section 5.2 summarizes the related work in this domain. Section 5.3 provides background information about web browsers, and discusses the nature of internal information that is available. Section 5.4 discusses different web browsing phases, followed by Section 5.5 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 5.6 and give a short summary in Section 5.7.

5.2 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 [10, 28, 31, 54, 81] 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 [83, 170, 168]. The works [83] and [168] 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 [170] 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 [154], 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 [171, 172], 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 QoS required by a mobile web application [169]. 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 by using events as in [169] (see Section 5.4).

Recent work has proposed to perform power management within the browser itself [11]. 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 can perform better power management.

Further, *RECON*, a model of the energy consumption of mobile web page loading [13] and a detailed analysis of the impact of the underlying platform architecture on web browser power consumption [124] have been presented. In [124], 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 [13] nor [124], while in this work we propose a *new* governor and compare it with the existing ones in Android. Our work extends the studies in [13], [124] and [170]. 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 [33, 35, 116], navigation and media streaming [39, 102]. 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 [29]. 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.

5.3 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.

5.3.1 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 see on the display and that we interact with. The page consists of static elements such as HTML and CSS, which 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 5.3 shows how the browser creates a usable web page. The 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 5.3. 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.

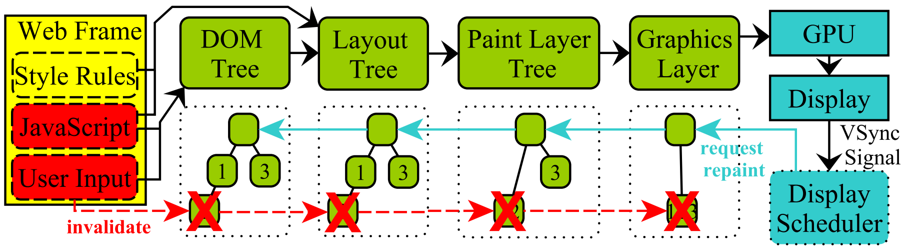


Figure 5.3: Updating a web page within the browser.

5.3.2 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 [46] for all of our experiments. The share of Chrome worldwide for mobiles and tablets is almost 60 % [106]. The second largest share is held 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. In the meantime, the user might want to interact with the web page. However, the main renderer thread is busy and cannot process the request. 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 the 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 5.4 and 5.5, we elaborate how this can be enabled and taken advantage of for the purpose of power management.

5.4 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.

5.4.1 User-Centric Performance Model RAIL

Within the Chrome browser, the user performance requirements are determined by the so-called user-centric *RAIL* model [47]. 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 5.1, that have been adapted from the HCI domain [108]. 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 playback. 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, 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.

5.4.2 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.

Table 5.1: Summary of the RAIL model [47].

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

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 needed 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's 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.

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 a 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 5.5 for details).

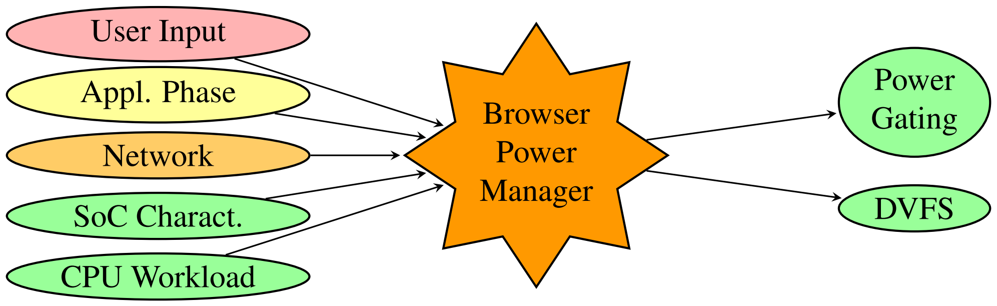


Figure 5.4: Browser governor information flow.

5.5 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.

5.5.1 Phase-aware Power Management Strategies

As depicted in Figure 5.4, 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 as much as possible using 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 % in 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 5.5. 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.

Idle

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

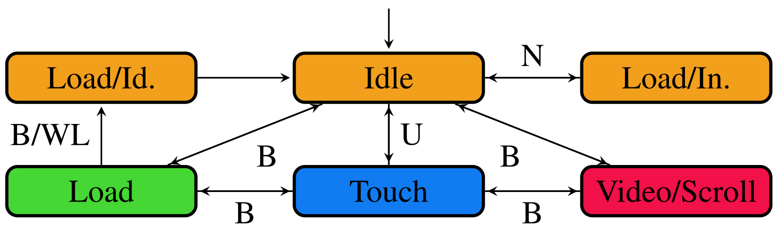


Figure 5.5: Phase state transition diagram of the browser governor. The transitions are based on user inputs (U), browser state changes (B), network traffic (N) or CPU workload (WL).

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 5.6 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 1.1.3. 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 deregistration 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.

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 5.3.2. 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. The scrolling state is usually entered from the *Touch* phase, so the A15 is turned on at the phase transition. 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. Once again, 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 the given 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.

5.5.2 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 5.6 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 shown in Figure 5.6 on the right. One may note that such modules already exist in other Android systems [153]. 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 the 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.

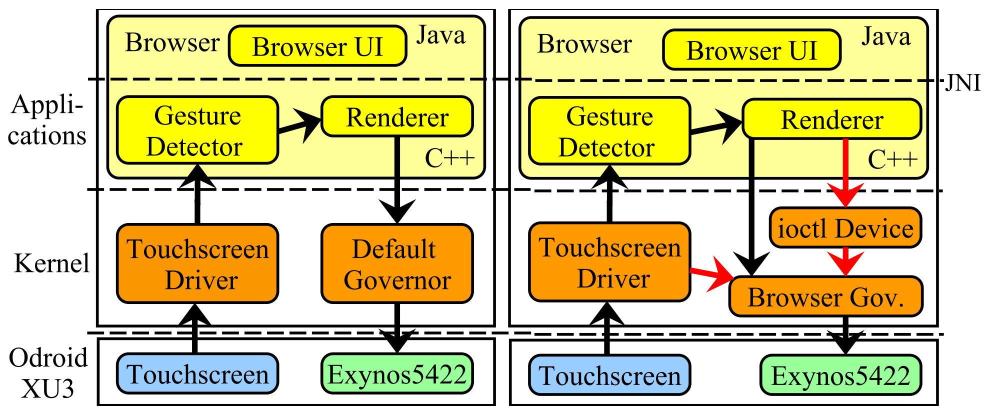


Figure 5.6: Regular touch event propagation (left) and browser governor (right) within the Android OS.

5.6 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 [4] and the Google Telemetry test suite [45].

5.6.1 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 5.7. Each bar has three sections corresponding to the power consumed by A7 (bottom), A15 (middle) and the GPU (top). In Figure 5.7 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 is 57.4 % (CNN) when compared to the interactive and 54.2 % (Amazon) when compared to the ondemand governor. The mean savings are 52.0 % and 51.5 %, respectively. Figure 5.7 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.

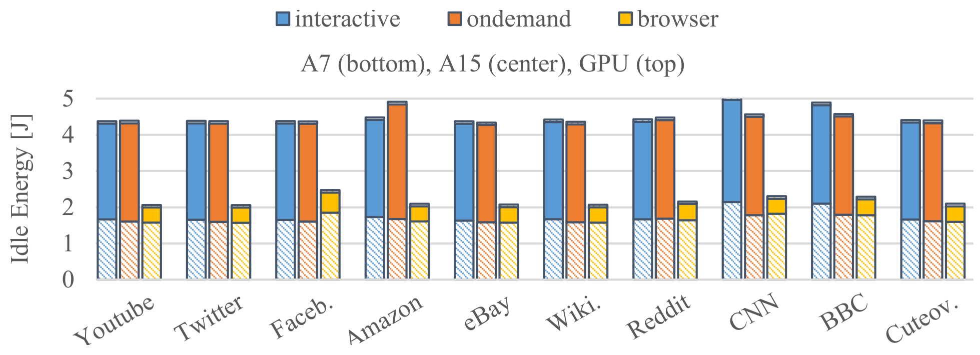


Figure 5.7: *Idle* phase energy consumption divided by consumers (A7, A15, GPU) for different governors.

5.6.2 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 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 5.8. 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.

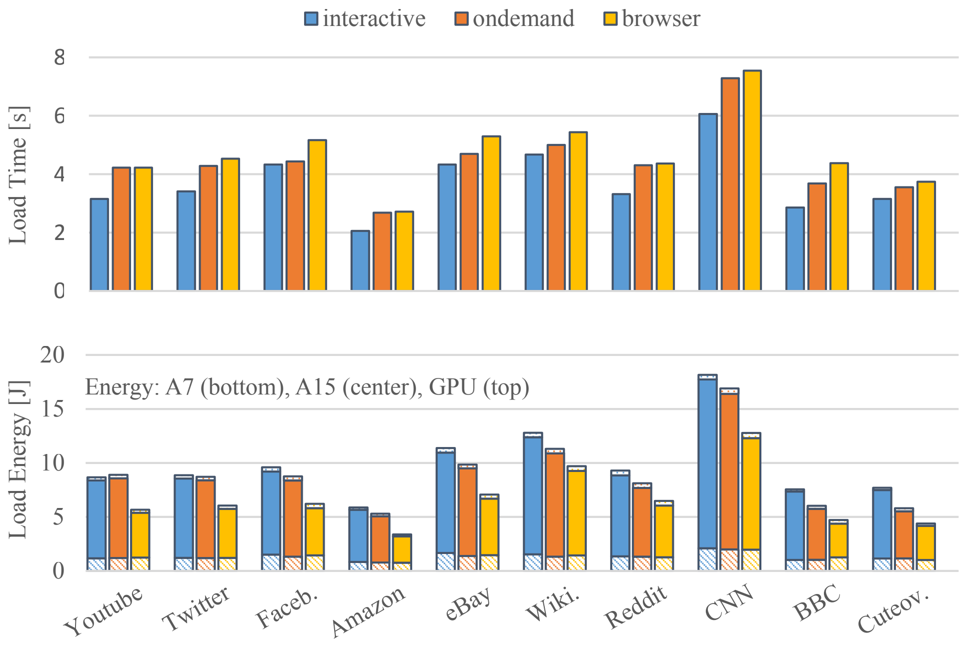


Figure 5.8: *Load* phase energy consumption (bottom) divided by consumers A7, A15 and GPU and load time (top).

5.6.3 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 workload to estimate the *Load/Idle* phase energy for the different governors. We defined the end of this phase as the time at which the A15 workload has been zero for more than 2 s. The results are shown in Figure 5.9. 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.

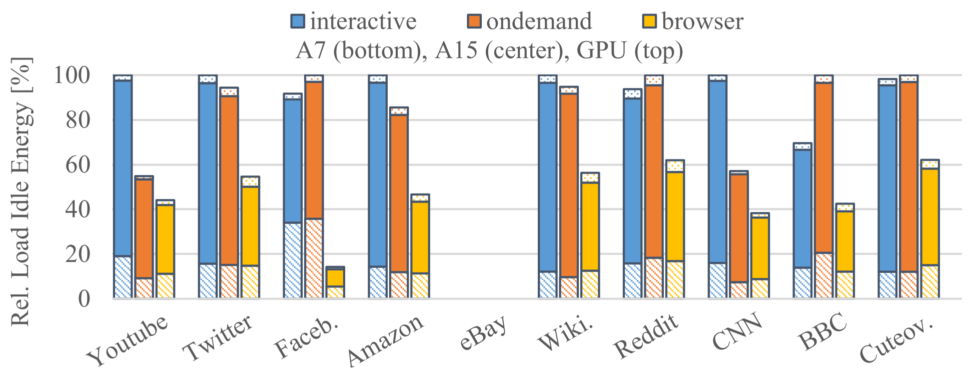


Figure 5.9: Normalized (see Section 5.6.3) background load energy consumption. The energy for eBay is zero.

5.6.4 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 5.10. As mentioned in Section 5.5.1, 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 5.10). 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 5.10), 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.

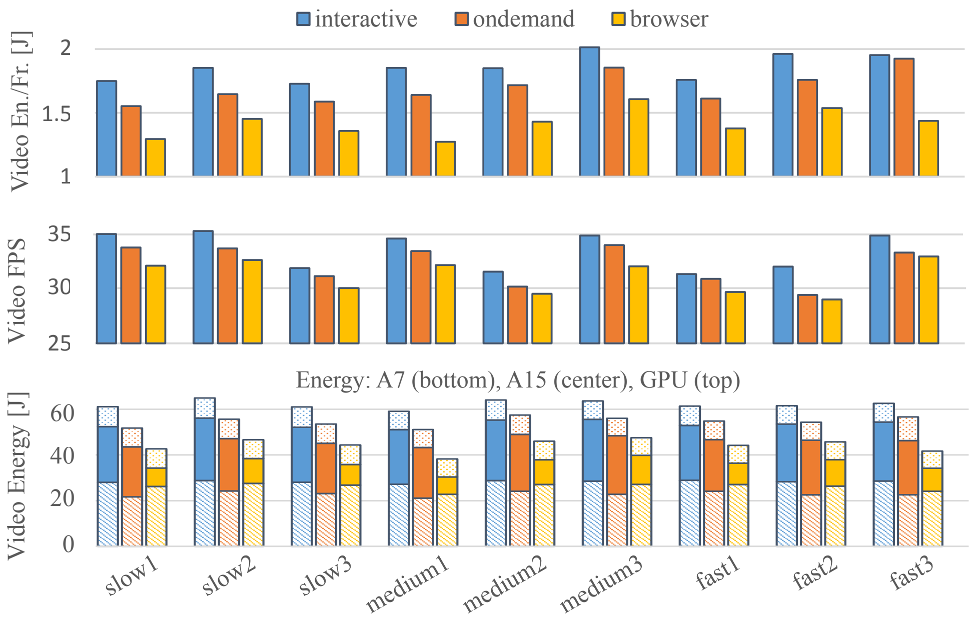


Figure 5.10: *Video* phase energy consumption over 60 s (bottom), frame rate (center), and energy per frame (top).

5.6.5 Scroll Phase

To evaluate the *Scroll* phase of the browser governor, we have recorded one long scroll gesture using the *reran* [41] tool and replayed it for all the web pages under test. As mentioned in Section 5.5.1, 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 5.11. 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, for example for Google+. Comparing the energy per frame values, the browser governor outperforms the interactive governor by 21 % and the ondemand governor by 1.7 % on average. 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.

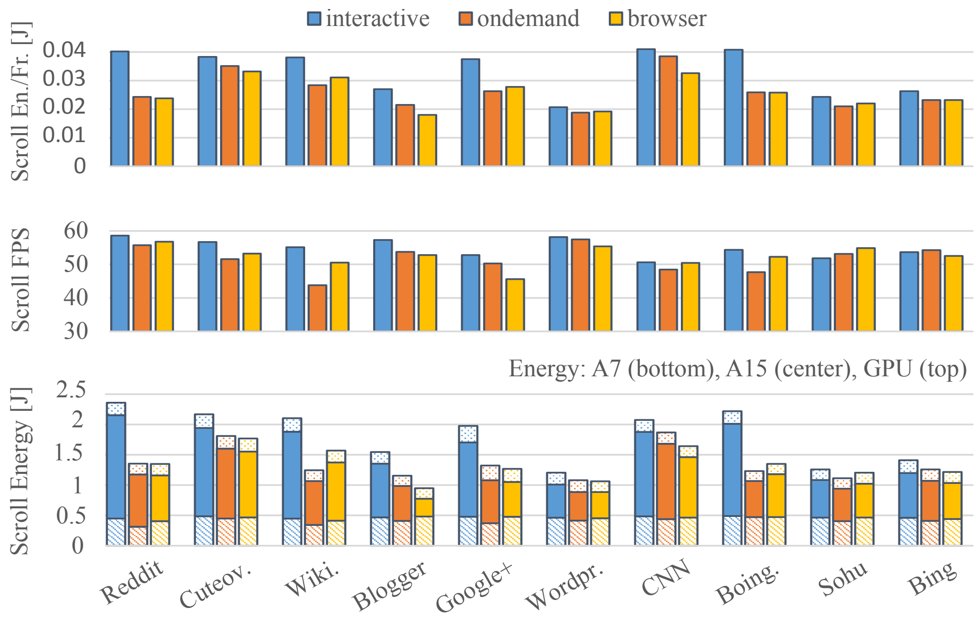


Figure 5.11: *Scroll* phase energy consumption (bottom), frame rate (center) and energy per frame (top).

5.7 Summary

In this chapter, we have introduced a phase-aware power manager for the Chrome browser. We identified that lack of coordination between the governor and the Chrome browser is a major hurdle in further reduction of the power consumption. 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 during power management. In particular, up to 57.4 % of used energy can be saved in idle phases, and 35 % in interactive/animation phases, without noticeable degradation in the performance. Moreover, we have investigated browsing phases such as loading and scrolling. In fact, our technique does a better job in achieving the target FPS because the frame rate is directly communicated from the browser. The results also show that the performance overhead of our technique – mainly related to the power gating overhead associated with A15 – is manageable. Our main goal is to build a generalized framework for Android governors, which is capable of incorporating information from different types of applications, not only the browser. We will explain how to exploit the results from the previous chapters to develop such a framework in the next chapter.

6 API for Power-Aware Application Design on Mobile Systems

In this chapter, we introduce the concept for an application-aware API that can be provided by the kernel for applications. As the implementation of such an API is burdensome in many ways, we will not only introduce the concept, but also discuss problems that arise with such an API and how these problems can be solved. We cover topics such as system integration, design hurdles and challenges for application developers.

6.1 Introduction

Until now, the power management has been considered a job to be handled by the OS or at the hardware layer, and energy-aware software development has not yet been in the main focus of the community [128]. For example, in case of Android systems, CPU schedulers and governors that reside in the kernel space perform thread allocation and determine the operating voltage and frequency of the CPU, have direct impact on the power consumption. Their decision is made based only on the information available within the kernel, which is mainly the CPU usage per core, without regards to the information available in the user space. If the usage of a CPU core increases above a certain threshold, the operating frequency is increased, and vice versa. Further, even if software developers consider the energy efficiency of their code as important, there is still a lack of tools for power-aware application design [128].

A key characteristic of applications running on mobile devices is that they are user experience-sensitive. For example, an application is expected to respond immediately to a touch event like scrolling or zooming, and maintain a certain frame rate during animations. On the other hand, there are background tasks that have little impact on the user experience. There is no need to rush executing work that is not perceivable by the user, but can increase the power consumption.

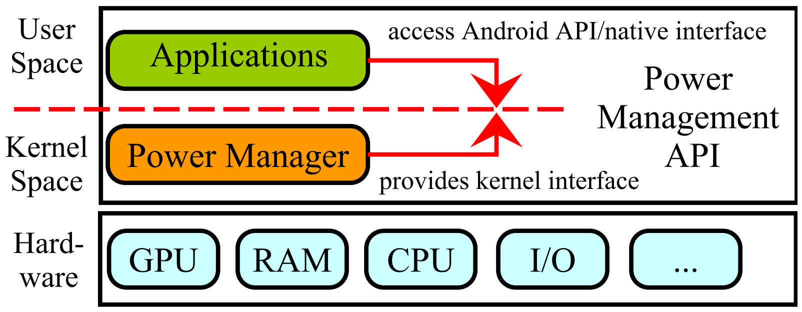


Figure 6.1: Possible Android System Infrastructure for a power-aware interface.

There have been prior works to reduce power consumption with minimal impact on the user experience for applications such as mobile web browsing [124, 169, 170], gaming [35, 116, 123], etc. Although they show the benefit of the interaction between the kernel and applications, these works are highly application-specific and require custom modifications to either kernel or user space applications.

The major drawback of the current Android software architecture is that the power management is done without regards to application-specific characteristics, which are not visible to the kernel. In the research community, however, the idea of creating a channel between an application and the OS has been considered in the past. The idea of energy-efficient software design for mobile devices is not a new one. The framework proposed in [39], adjusts *fidelity*, which allows leveraging trade-off between the energy consumption and user-experience in multimedia and maps applications. This work was continued later and adopted to newer platforms [102]. It is orthogonal to ours as it does not cover specific power management strategies for the CPU. Numerous other techniques exist regarding power-aware code design, amongst several others are [84, 87, 88, 112, 115]. Nevertheless, the works are still application-specific and therefore have limitations in expandability to other applications or place too much responsibility on the application developers.

In this chapter, we show that there is a lack of support from the operating system to enable an application to pass over user requirements or application-specific information to the kernel. Then, we provide two use cases, mobile gaming and web browsing, to motivate an API between the user space and the power managers in the kernel space for better CPU power management, as depicted in Figure 6.1. Such an API should be general enough to be applicable to a wide variety of applications. We believe that such a generalized framework allowing coordination of applications and the operating system, would encourage incorporating previously proposed power reduction techniques to real products.

6.2 API for Power Management

In this section, we propose an API for power-aware application design in Android. We introduce two examples, namely mobile games and web browsing, where sharing information between application and governor results in substantial power savings and then we derive a generalized

API based on these examples. Finally, we evaluate the consequences for developers when establishing such an API.

6.2.1 Application-Aware Power Management

We use two examples to explain which information can be passed between kernel and applications. Games show how target frame rate and timing deadlines can be exploited for interactive workloads. The browser work emphasizes the importance of proper workload prioritization, which can only be extracted from the application itself.

Game Applications

Mobile games are very popular, but power hungry and computation intensive applications. The complexity of modern games' graphics, physics and AI is constantly increasing, what poses high workloads on the system. Research has shown that the resource demand of games is state- and frame-bound [33, 35, 123]. State-bound means that workload characteristics change significantly according to states. For example, in a *loading* state, the workload is heavily memory-bound and graphical effects are often minimized. On the contrary, graphical effects and their computations are dominant during the actual *playing* state. For loading, the frame rate can be low but the loading should finish as fast as possible. For the game play, the target FPS value is normally 30 or 60 to maintain a good user experience.

From the gaming example, we identify two classes of information that can be passed from the applications to the governor, *frame rates* and *deadlines*. A simple approach to control the power consumption is to pass the target frame rate and the achieved frame rate to the governor [117, 118]. The governor can regulate the frequency based on the discrepancy of the achieved and the target frame rate. Further, the frame rate can be considered a deadline [35, 123]. The goal of the power management strategy is to calculate the frame (physics, etc.) within a particular time slot (e.g., 16.67 ms for 60 FPS). Predictive strategies can be applied to calculate the workload of the next frame based on the past workloads. This information can be used to find the appropriate frequency level for the particular frame that just fulfills the frame's resource requirements. Applying this strategy, frequency overshoots can be avoided during high workload phases in games and power can be saved. Recent gaming works that implements such strategies report up to 43.2 % energy savings for dual-core CPUs [35] and on average 41.9 % for HMP architectures [123] compared to the Android default interactive governor.

The work on game power management shows the importance of sharing information between the application and the governor. Providing the frame rate as a computation deadline results in large power savings as the frequency can be dynamically adjusted. Without this information, the governor cannot estimate the time that it takes to complete the frame calculation. Consequently, it tries to finish this job as fast as possible, which results in power wastage.

Mobile Web Browsing

The web browser is another popular smartphone application [44] and we have shown that there is a large potential for power savings in mobile web browsing [124, 126]. Similar to gaming, browsing can be divided into different states such as loading a page, scrolling, etc. The so-called

RAIL model [47] summarizes the performance requirements for different states. As explainted above, RAIL defines the *response* time for actions, the time to display *animations*, *idle* state behavior and the maximum *loading* time for mobile and desktop devices. Moreover, there is application-specific data (e.g., loading state) readily available within the browser.

Based on the RAIL model, we have defined information that can be passed from the application to the governor [126]. For example, maximum response and loading times can be handled as *deadlines* if the workload can be determined. Otherwise, both actions can be treated as high *priority* actions that need to be completed as fast as possible. As described in Section 5.1, the state of the current action can be used to distinguish between high workloads that are critical (foreground load) and others that are not necessarily relevant for a good user experience (background load). Such information can solely be provided by the application and by no other entity in the system. Further, the animation action requires a minimum FPS value to be maintained. Scrolling or zooming actions, but also videos playback are considered animations. Similar to gaming, the FPS value can be used as a *deadline* or as a target *frame rate*.

As we have already shown in Chapter 5, the power saving potential for the web browser is very high, when application-specific information is utilized. In particular, the information about the loading state to distinguish between foreground and background loading is very promising. The measurement results in Section 5.6 show, that power savings up to 50 % are possible if considering the application state. As mentioned above, the application states can be translated to the *priority* of the workloads and a corresponding power management strategy. This is of particular importance for hardware platforms with a large discrepancy between the standard operating mode (e.g., mostly A7 for the Odroid-XU3) and a very power consuming high-performance mode (e.g., highest frequency of the A15 for the Odroid-XU3).

6.2.2 API between Applications and Kernel

As shown in the previous sections, there is a set of common information among Android applications that can be used for power management. Gaming and browsing already capture a wide range of application types. For example, social media applications such as Facebook and Twitter can be compared to scrolling through a web page. Other applications, e.g., navigation, are graphics intensive such as games. In general, *all* user interactive applications need to maintain a *frame rate*, have high *priority* phases such as displaying incoming messages or other timing

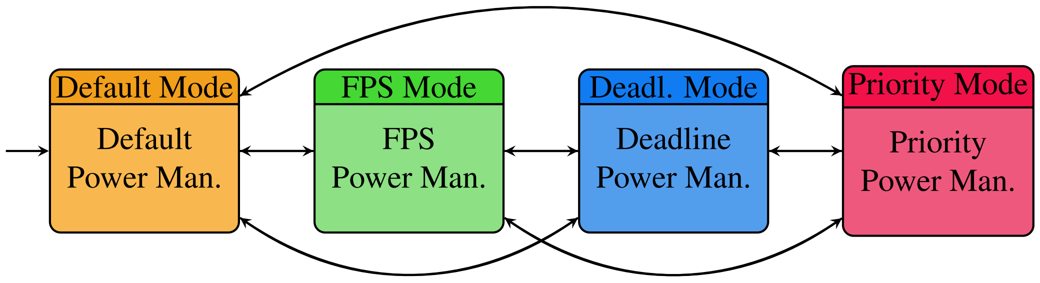


Figure 6.2: Power manager with application-aware strategies.

constraints such as *deadlines*. Hence, we suggest a power manager that can switch between different power management strategies, namely *modes*, as depicted in Figure 6.2. The mode can be changed by the developer on demand. Besides one default mode (e.g., an Android default governor), the developer can choose the mode based on the state of the application and needs to provide mode-specific parameters as described in the following.

Frame rate

The examples in the previous section show that the FPS value can efficiently be exploited for power management for interactive applications or videos. As described, the frame rate can be seen as a target value that can be provided by any application that performs frame-based calculations. For example, there are games that run at 60 FPS, but nowadays there are also many games that target 30 FPS. Some applications might even target a lower frame rate. If the governor knew the target value, it could adjust the exact frequency level that is needed by the application. This is not possible by using workload information only. As of now, there is no possibility to acquire the current frame rate within the kernel without input from the application layer. The frame rate information within the kernel is obtainable from the GPU source code, that it is usually closed-source. Hence, the current FPS value has to be periodically passed from the application to the kernel to adjust the control loop. However, a GPU API in the kernel space could even reduce the communication overhead between the Android layers.

Deadlines

Deadline information could be useful in combination with workload estimations. In many applications, there exists a temporal correlation of frame-based workloads, which allows quite accurate workload estimations inside the governor [33, 35, 123]. For games, we can consider the VSync of the display as a deadline for processing frames. One of the major challenges when implementing such a strategy is coordinating multiple deadlines for different applications or tasks within the system. If the workload is not known, the frequency of the CPU cannot be estimated. If there are multiple deadlines for different applications or task within the system, a deadline based scheduling algorithm has to be implemented within the kernel.

Workload Priority

From the browser power management work we have learned that workloads can be prioritized differently and, hence, require different power management techniques. For example, we can distinguish between foreground and background load, because the browser keeps track of the web page loading state. Generally, developers are aware of performance-critical sections within their application. Such sections can be loading, displaying data or involve complex algorithms, such as software encoding or decoding of data streams. The priority of those critical sections with regards to user perception could be given to the power manager as a hint. This would inform the power manager not to restrict any resources when that part of the application is running. In the browser example, the foreground load can be marked as *high* priority while the background load can have a *low* or a *default* priority level. As a consequence, the power manager can tune its strategy as needed. However, it should be aware, e.g., in case of HMP platforms, that the user responsiveness is not degraded by limiting the available resources. Hence, the power manager should be aware of the workload *and* the underlying hardware platform to make a proper decision. Generally, the characteristics of the application workloads were not obscure anymore, and the power manager could perform better power management.

6.2.3 Implementation Issues and Challenges

Providing a power-aware API between the application and the kernel layer requires modifications to the Android OS. Moreover, we describe which challenges such an API poses for application developers. While every new feature brings more complexity to the system, our initial studies, and numerous previous work point out that the potential power savings surely outweigh the overhead.

System Integration

As shown in Figure 6.1, we need to establish a *communication channel* between the application and the kernel. First, the power manager needs to provide an interface accessible from the user space. This can be realized using so-called *ioctl* devices that are created within the kernel itself. Second, the current Android API needs to be extended such that application developers can directly access the power manager within their (Java) source code. Third, the API needs to be designed with respect to *downward compatibility*. Hence, existing or newly developed applications that do not make use of the power management API should not be affected by the feature. These applications could be run using the default mode of the power manager.

Hardware Architecture Support

While the Android API abstracts away the underlying hardware platform, the power manager within the kernel needs to be *platform-aware* to achieve the best possible power savings. For example, if the workload is defined as a low-priority workload, the actions of the power manager could vary depending on the hardware, e.g., powering down the big CPU for HMP platforms, lowering the frequency for single-CPU systems, etc. Moreover, a scenario may occur, where an application developer makes use of the API but the different power management modes are not supported by the kernel. This should be resolved and abstracted away from the application developers.

Power Manager Design Hurdles

The implementation of the power manager within the kernel is not trivial. Hence, we address some major design issues that have to be treated such that the API would bring the intended benefits. First, the *prioritization* among the modes needs to be defined in case that there are two tasks within the system that request different modes from the API. For example, high priority workload should not be affected by frequency reductions for deadlines or frame rates. Further, if there is a conflict between deadline and frame rate modes, the power manager should ensure that the requirements of both modes are fulfilled.

Second, it is important to define what shall be done in case the API is *misused*. One type of misuse is applications requesting too many resources although there is only little work that needs to be done. This can result in an unnecessary high power consumption and consequently in faster battery drainage. The situation can be mitigated by designing power management strategies that monitor the workload in addition to the mode parameter. If the workload is low although the workload priority is set to high, there will be no performance benefit from keeping the CPU frequency at its maximum. However, misusing the API can also mean that an application has a large resource demand but restricts its own resources through the API. This situation is not as easy to resolve as the former one because saving power is the exact purpose of the power manager. However, extensive tests of the application should reveal situations where the mode is not set correctly. For example, an Android debug option that shows the mode and the workload on the screen could be implemented.

Challenges for Application Developers

As discussed in [128], software developers currently lack the knowledge and tools to design applications with respect to power consumption reduction. On the one hand, software development should be as easy as possible. Hence, developers should worry about as few problems as possible. On the other hand, power consumption is such an important issue in mobile devices, nowadays, that it deserves more attention from the developers and the research community. For the power-aware API, general guidelines can help application developers to choose the correct mode. For example, interactive applications usually target a specific frame rate. Moreover, fast loading and rendering action usually do not require a particular frame rate but have a high workload priority. There could also exist background tasks that pose a high workload on the system but can be completed over a long period of time that could be used as a deadline. Tools that visualize the workload of an application and assist in identifying the mode that should be passed to the API would be even more helpful to the developers.

Security and Misuse

One of the major issues when adding new features to a system nowadays is security. The developer needs to make sure that his application or feature will not cause any damage to the user and the data stored on the device. The API that we are proposing could be misused as described in the previous section. For example, the system could become very slow due to wrongly requesting an idle mode. On the other hand, the battery could drain fast, if a high priority mode is requested when the idle mode could suffice. However, implemented wisely it should not cause any malicious threats to the system that crash the complete system.

First, the API should only take a predefined sequence of commands. All other commands should be ignored as errors. Second, the API should only provide tuning parameters to the power manager. This means that the power manager has to anyway take the workload of the system into account to perform proper power management. Even if the application told the power manager that critical workload should be handled, it would not raise the CPU frequency in case there is no actual workload. Still, an application could tell the governor that there is no critical workload although this is not true. That application is obviously not designed correctly if the workload is caused by the application itself. However, the power manager needs to make sure that there is no other application that is running concurrently and could be starved.

6.3 Summary

Despite a considerable amount of prior work on power management for applications running on mobile devices, introduction of such techniques into real products has been sluggish. This is due to, on the one hand, lack of standardized means in Android systems to pass the application-specific information and user requirements to the operating system, and on the other hand, customization efforts required for implementing state-of-the-art power management techniques. In this chapter, we observed two applications, games and web browsers, to identify information that are useful for power management done in the operating system, and propose an API for energy-efficient application development. We also discussed implementation details and acceptance issues within the community. As power consumption of mobile devices is of ubiquitous importance, efforts to push power saving techniques to the general awareness are of major importance. The user space applications may simply notify the desired frame rate, deadlines, and priorities for workloads to the kernel explicitly to let the operating system perform more effective power management. We think that the proposed framework is general enough to be accepted by the mobile software development community.

7 Conclusions and Future Work

In this thesis, we investigated the power consumption of different interactive Android applications, namely games and web browsers, and proposed a generalized power management API between applications and the kernel based on our observations. The state-of-the-art power managers in the Android system, the interactive and the ondemand CPU governors, already perform a decent job. However, there is a lot of potential to save power, especially on modern big.LITTLE platforms, which feature Heterogeneous Multi-Processing.

7.1 Summary

As we have shown, the full potential of HMP SoCs cannot be fully exploited by the Android default governors. Instead, SoC-specific characteristics should be considered when designing a power manager, taking into account, for example, that one CPU can be power gated in case the chip features multiple CPUs. We have shown, that such an approach results in significant power savings and also verified our approach in a user study in our work on game power management. Moreover, we have looked at the power saving potential of HMP platforms for web browsers and found that a considerable amount of power can be saved if only little a performance was sacrificed. We took the results of this work as a cue and implemented our own power manager that takes into account not only the underlying hardware platform but also information from the browser application itself. Based on our observations from the browser governor and the game-specific governor, we could derive more generalized states that are common for most Android applications. Consequently, we have concluded the work by proposing a general API for Android applications that is not bound to a specific application. The main contributions of this work are summarized as follows:

Frame- and Thred-based Workload Prediction for Games

We introduced a new prediction frame work for games that takes into account not only the workload of the complete system, but looks into the specific threads spawned by the application. Based on the previous workloads of the threads, the power manager decides what kind of prediction strategy to apply for a particular thread. We have tested several predictors and have found the following most effective: The thread workload can be part of a time series where the current workload depends on the workloads of the preceding frames. Such workloads can be predicted using a WMA predictor. Other threads are executed periodically. This can be detected by an ACR predictor. We found that a combination of the WMA and ACR predictors yields the best results.

The GameOptimized Governor for an HMP Platform

To perform power management on the HMP platform, we have created a software and a hardware model. The software model corresponds to the frame- and thread-based workload prediction strategy of the governor. The hardware model has to be adopted to the HMP platform – namely the available CPU cores and frequencies. We use a so-called migration factor to compare the capacity of big and little cores. The power manager tries to fill up the cores based on the available capacity and the predicted workload of the threads. With the GameOptimized governor, we can save on average 41.9 % of total energy consumption while the user experience is still rated good and very good.

Web Browser Workload and Power Measurement Setup

While the Odroid-XU3 board provides sensors to measure the power consumption of the CPU, there is no standardized way to measure the power consumption of an application divided by its threads. We have developed a software setup, that gathers information from the built-in power sensors, the running browser processes and the Chrome:trace tool. This setup enables us to calculate the the power consumption per thread and even per browser task and function based on the collected information.

Web Browser Workload Characterization on an HMP Platform

We performed a detailed characterization of the browser processes and in particular the rendering process on our experimental platform, the Odroid-XU3 board. We showed that the time the browser spends on the A7 and the A15 is almost equal, however, the energy consumption of the A15 is between four and six times higher. Moreover, we showed that the CrRendererMain process contributes most to the time and energy consumption of the browser power consumption. By further analyzing the CrRendererMain process and looking into JavaScript-related functions, we found that up to 40 % of the time is spent in parsing and compiling the JavaScript code, rather than executing it.

Web Browser Power-Performance Analysis

We performed a power versus performance analysis of the A15 cores, where we studied the effects of frequency capping, thread-to-core allocation and power gating on the loading times of web pages. We found that sacrificing only a little bit of performance results in significant power saving. This is especially noticeable for power gating the A15 that reduces the idle power of the A15 by 85 %. However, the effect of reducing the A15 frequency is also remarkable. For example, when running the A15 at the minimum (1.2 GHz) instead of the maximum (2.0 GHz) frequency, the power consumption can be decreased by 34.6 % while the loading time increases by only 16.7 %.

Phase-Aware Web Browser Power Management on HMP Platforms

We took the results from the characterization and analysis to implement our own browser-aware CPU frequency governor. Therefore, we divided the browsing procedure into so-called phases that exhibit different performance characteristics and therefore require different power management techniques. We introduced phases such as idle, load/idle, load, scrolling and video. For all phases, we implemented different power management strategies within the kernel. To identify the phases, we also had to create a link between the application and the kernel layer that enables us to pass information from the browser to the power manager. Our results show that we can save up to 57.4 % of energy for idle phases where we can exploit the power gating of the A15. Moreover, we can save up to 35 % of energy for interactive and animation phases.

API for Power-Aware Application Design on Mobile Systems

Based on our findings in the previous works, our main goal is to propose a power manager that is not only capable of saving power for one particular application, such as the web browser or games, but that can be applied to all kinds of Android applications. Therefore, we suggested that Android applications can be divided into different states that are similar across all applications. Such states are, for example, interactive states that require resources for short-term peak performance. Moreover, there are states where a particular frame rate needs to be targeted, e.g., in videos or animations. In some applications, such as games, the frame-based workload can be predicted and the deadline for the execution of the task is known – this can be exploited for power management. Based on these states, we have proposed an API that allows the application developer to pick a state for his or her particular application and to provide the necessary information directly to the governor.

7.2 Future Work

The techniques presented in this thesis show that there is a significant amount of potential for power savings in mobile Android systems. However, this thesis is only able to target a small amount of problems present in such a broad field of research, such as Android power management. In the following, we outline some follow-up challenges that arose from our work.

Workload Predictors for Mobile Games

We have implemented a workload predictor that exploits the time correlation between neighboring frames and also the autocorrelation for periodic workloads. Moreover, many other workload predictors were proposed in previous works, e.g., PID predictors, LMS predictors and ARMA models. While all these approaches outperform the Android default governors, it was shown that there is still significant room for improvement in terms of power savings [32]. Based on these observations, the exploration in the area of workload predictors becomes interesting.

More involved predictors could be implemented, for example, non-linear predictors or even neural networks-based predictors or machine learning techniques. Of course, such predictors could also take the GPU and other hardware components into account.

Estimating the Limits of CPU Power Management for Mobile Games on HMP Platforms

The work in [32] suggests that there is significant room for improvement in terms of CPU power consumption for mobile games. However, the work does not look at an HMP platform but an older platform, which only features a dual-core CPU. Nowadays, multi-core SoCs have become state-of-the-art and it seems natural to extend the work in [32] to a more modern hardware. We assume that the optimal power savings that can be achieved on an HMP platform would even outperform the optimal savings on a simpler platform. This is due to more degrees of freedom of an HMP platforms that include power gating and a larger number of CPU frequencies that result in higher saving when applying DVFS.

Frame Rate Adaption for Power Management Based on Scrolling Speed

From the browser example, we have seen that scrolling is very sensitive to the current frame rate. However, we have not tested the effects of different scrolling speeds on the power consumption. In general, we can assume that different scrolling speeds also imply different perception ability from the user. For example, when the user is scrolling very fast, then the screen appears blurry. Consequently, we could lower the frame rate as there will be no deterioration to an animation. On the other hand, when the user is scrolling very slowly, a low frame rate usually does not affect the user perception, either [27]. However, by lowering the frame rate and rendering less frames to the screen, a significant amount of power could be saved. We think that exploring such strategies and verifying them in a user study could lead to a new approach for power management when scrolling, which is a very common gesture not only on mobile devices but also on desktop computers.

Enhanced Browser Governor and User Study

We have implemented and tested the browser governor on our Odroid-XU3 platform. While the results are overall very promising, there is still potential for improvement of the governor. For example, we found that power gating the A15 enables us to save a lot of power during idle states. However, turning the A15 back on comes with some time overhead, that is slightly noticeable in the results. We believe that the algorithms can be fine-tuned to minimize this overhead even more. Moreover, it is important to verify power management concepts in terms of usability not only by benchmarks but also in real user studies. Should a user study confirm that the user perception of the browser governor is comparable to the interactive and ondemand governors, then this would be a great step towards implementing A15 power gating for running applications.

Power Management beyond the CPU

In this work, we have focused our studies on CPU power management, only. While this is a broad field of research, there exist more components that contribute significantly towards the power consumption in a mobile device. We have already introduced the display and the wireless link in Chapter 2. We believe that combining power management strategies for all of these components would result in even larger power savings than performing power management for each component individually. For example, scrolling speed-aware frame rate adaption could not only affect the CPU power consumption, but also might be combined with dimming the display - or even parts of the display for OLED technologies. Also, our work on the power-aware API gives cues for power management strategies for individual components, even if not considered in combination with each other. As most hardware components feature different power states, nowadays, we suggest to define not only a power management strategy for the CPU, but also for the display and the wireless link based on the requirement of the particular state. While such states are partly implemented, especially for the display, we think that there is significant potential for improvement.

Implementation and Verification of the API for Power-Aware Application Design

We have proposed an API for power-aware application design in Chapter 6. However, we have not yet implemented the API and verified the concept. Besides the actual implementation for our Odroid-XU3 platform, there are a lot more interesting problems that arise in the context of power-aware programming. First, there is the problem of designing such an API not only for one platform but such that it is applicable for many platforms. This requires a good balance between generalization and specification to the hardware. Moreover, cases where the API is not used or not supported need to be considered. Second, power-aware programming has neither been in the center of attention in the industry nor in academia. There is a lack of tools, concepts and general guidelines on how to create applications in such a way that they consume less power. We see a lot of potential to 1) bring the awareness of power consumption to the developers and 2) create tools that will help the developers programming applications that consume less power. Such an awareness could be a great step towards developing power-aware applications in the future.

List of Figures

List of Tables

Acronyms

ACR	Autocorrelation Predictor.
AI	Artificial Intelligence.
AOSP	Android Open Source Project.
API	Application Programming Interface.
CPU	Central Processing Unit.
CSS	Cascading Style Sheets.
DOM	Document Object Model.
DVFS	Dynamic Voltage and Frequency Scaling.
DVS	Dynamic Voltage Scaling.
FPS	Frames-per-Second.
GPU	Graphics Processing Unit.
GSM	Global System for Mobile Communications.
GTA	Grand Theft Auto.
HCI	Human-Computer Interaction.
HMP	Heterogeneous Multi-Processing.
HTML	Hypertext Markup Language.
HTWE	Hilbert Transform-based Workload Estimation.
HVS	Human Visual System.
IC	Instruction Count.
IDCT	Inverse Discrete Cosine Transformation.
IPC	Instructions per Cycle.
JDK	Java Development Kit.
JIT	Just–In-Time.
JPEG	Joint Photographic Experts Group.
LCD	Liquid Crystal Display.
LMS	Linear Mean Square.

LTE	Long Term Evolution.
MC	Motion Compensation.
MMU	Memory Management Unit.
MPEG	Moving Picture Experts Group.
MPSoC	Multi-Processing System on Chip.
NDK	Native Development Kit.
NLMS	Normalized Linear Mean Square.
OLED	Organic Light-Emitting Diode.
OS	Operating System.
PCI	Peripheral Component Interconnect.
PCIe	Peripheral Component Interconnect Express.
PDA	Personal Digital Assistent.
PID	Proportional Integral Derivative.
PNG	Portable Network Graphics.
QoS	Quality of Service.
SNR	Signal-to-Noise Ratio.
SoC	System on Chip.
SRT	Soft Real-Time.
SSIM	Structural Similarity Index.
TCP	Transmission Control Protocol.
UTMS	Universal Mobile Telecommunications System.
VLD	Variable Length Decoding.
VoIP	Voice over IP.
VSync	Vertical Synchronization.
WMA	Weighted Moving Average.

Bibliography

[1] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless wake-ups revisited: Energy management for voip over wi-fi smartphones. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2007.

[2] E. Ahmad and B. Shihada. Green smartphone GPUs: Optimizing energy consumption using GPUFreq scaling governors. In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2015.

[3] E. Akyol and M. van der Schaar. Complexity model based proactive dynamic voltage scaling for video decoding systems. *IEEE Transactions on Multimedia (TMM)*, 9(7):1475–1492, 2007.

[4] Alexa Internet, Inc. The top 500 sites on the web. `http://www.alexa.com/topsites`, 2016.

[5] B. Anand, L. Kecen, and A. L. Ananda. Parvai — hvs aware adaptive display power management for mobile games. In *International Conference on Mobile Computing and Ubiquitous Networking (ICMU)*, Jan 2014.

[6] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[7] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[8] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *SIGCOMM Conference on Internet Measurement (IMC)*, 2009.

[9] H. M. K. G. Bandara and H. A. Caldera. Towards optimising wi-fi energy consumption in mobile phones: A data driven approach. In *International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2015.

[10] Brave Software Inc. Brave. `https://brave.com/`, 2017.

[11] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking energy performance trade-off in mobile web page loading. *GetMobile: Mobile Computing and Communications*, 20(2):14–26, 2016.

[12] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Characterizing web page complexity and its impact. *IEEE/ACM Transactions on Networking (TON)*, 22(3):943–956, 2014.

[13] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the energy consumption of the mobile page load. *ACM on Measurement and Analysis of Computing Systems*, 1(1):6:1–6:25, 2017.

[14] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Conference on USENIX Annual Technical Conference (USENIXATC)*, 2010.

[15] M. Chan. cpufreq: interactive: New 'interactive' governor. `https://lwn.net/Articles/662209/`, 2015.

[16] N. Chang, I. Choi, and H. Shim. Dls: dynamic backlight luminance scaling of liquid crystal display. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):837–846, Aug 2004.

[17] B. Chen, X. Li, X. Zhou, T. Liu, and Z. Zhu. Towards energy optimization based on delay-sensitive traffic for wifi network. In *International Conference on Ubiquitous Intelligence and Computing (UIC)*, 2014.

[18] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[19] X. Chen, J. Zeng, Y. Chen, W. Zhang, and H. Li. Fine-grained dynamic voltage scaling on oled display. In *Asia and South Pacific Design Automation Conference*, 2012.

[20] X. Chen, J. Zheng, Y. Chen, M. Zhao, and C. J. Xue. Quality-retaining oled dynamic voltage scaling for video streaming applications on mobile devices. In *Annual Design Automation Conference (DAC)*, 2012.

[21] Y. Chen, X. Chen, M. Zhao, and C. J. Xue. Mobile devices user: The subscriber and also the publisher of real-time oled display power management plan. In *International Conference on Computer-Aided Design (ICCAD)*, 2012.

[22] W.-C. Cheng and M. Pedram. Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling. *IEEE Transactions on Consumer Electronics*, 50(1):25–32, Feb 2004.

[23] Z. Cheng, X. Li, B. Sun, C. Gao, and J. Song. Automatic frame rate-based DVFS of game. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015.

[24] K. Choi, K. Dantu, W.-C. Cheng, M. Pedram, M. Pedram, M. Pedram, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *International Conference on Computer-aided Design (ICCAD)*, 2002.

[25] H. Chung, M. Kang, and H.-D. Cho. Heterogeneous multi-processing solution of Exynos 5 Octa with ARM® big.LITTLE technology. In *Samsung White Paper*, 2012.

[26] K. T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games. *Multimedia Systems*, 13(1):3–17, 2007.

[27] M. Claypool, K. Claypool, and F. Damaa. The effects of frame rate and resolution on users playing first person shooter games. In *Multimedia Computing and Networking*, 2006.

[28] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. *SIGPLAN Notices*, 50(11):105–117, 2015.

[29] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[30] S. K. Datta, C. Bonnet, and N. Nikaein. Android power management: Current and future trends. In *Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT)*, 2012.

[31] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. Idle time garbage collection scheduling. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[32] B. Dietich, N. Peters, S. Park, and S. Chakraborty. Estimating the limits of cpu power management for mobile games. In *International Conference on Computer Design (ICCD)*, pages 1–8, 2017.

[33] 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.

[34] B. Dietrich and S. Chakraborty. Forget the battery, let's play games! In *Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2014.

[35] B. Dietrich and S. Chakraborty. Lightweight graphics instrumentation for game state-specific power management in Android. *Multimedia Systems*, 20(5):563 – 578, 2014.

[36] B. Dietrich, D. Goswami, S. Chakraborty, A. Guha, and M. Gries. Time series characterization of gaming workload for runtime power management. *IEEE Transactions on Computers (TOCS)*, 64(1):260–273, 2015.

[37] B. Dietrich, S. Nunna, D. Goswami, S. Chakraborty, and M. Gries. LMS-based low-complexity game workload prediction for DVFS. In *International Conference on Computer Design (ICCD)*, 2010.

[38] M. Dong, Y.-S. K. Choi, and L. Zhong. Power-saving color transformation of mobile graphical user interfaces on oled-based displays. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[39] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, 1999.

[40] T. Garsiel. How browsers work. http://taligarsiel.com/ Projects/howbrowserswork1.htm, 2009.

[41] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *International Conference on Software Engineering (ICSE)*, 2013.

[42] Google Inc. Chrome V8. `https://developers.google.com/v8/`, 2015.

[43] Google Inc. Nexus 5X. `https://www.google.com/nexus/5x/`, 2015.

[44] Google Inc. There's an app for that...the browser, 2015.

[45] Google Inc. Catapult. `https://chromium.googlesource.com/catapult/`, 2017.

[46] Google Inc. Chrome. `https://www.google.com/intl/en/chrome/browser/desktop/index.html`, 2017.

[47] Google Inc. The RAIL Performance Model. `https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail`, 2018.

[48] Google Inside AdWords. Building for the next moment. `http://adwords.blogspot.co.uk/2015/05/building-for-next-moment.html`, 2015.

[49] K. Grochla and P. Foremski. Demo: Wireless link selection on smartphone: Throughput vs battery drain. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2015.

[50] Y. Gu and S. Chakraborty. Control theory-based DVS for interactive 3d games. In *Annual Design Automation Conference (DAC)*, 2008.

[51] Y. Gu and S. Chakraborty. A hybrid dvs scheme for interactive 3d games. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[52] Y. Gu and S. Chakraborty. Power management of interactive 3D games using frame structures. In *International Conference on VLSI Design (VLSID)*, 2008.

[53] Y. Gu, S. Chakraborty, and W. T. Ooi. Games are up for dvfs. In *Annual Design Automation Conference (DAC)*, 2006.

[54] S. K. Gudla, J. K. Sahoo, A. Singh, J. Bose, and N. Ahamed. Framework to improve the web application launch time. In *International Conference on Mobile Services (MS)*, 2016.

[55] J. Gui, D. Li, M. Wan, and W. G. J. Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *International Workshop on Green and Sustainable Software (GREENS)*, 2016.

[56] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *International Conference on Software Engineering (ICSE)*, 2015.

[57] B. Han, F. Qian, S. Hao, and L. Ji. An anatomy of mobile web performance over multipath TCP. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015.

[58] B. Han, F. Qian, and L. Ji. When should we surf the mobile web using both wifi and cellular? In *Proceedings of the Workshop on All Things Cellular: Operations, Applications and Challenges (ATC)*, 2016.

[59] Hardkernel co., Ltd. Odroid-XU3. `http://www.hardkernel.com`, 2015.

[60] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, 2014.

[61] K. Ho, C. King, B. Das, and Y. Chang. Characterizing display qos based on frame dropping for power management of interactive applications on smartphones. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.

[62] Z. Hu, Y.-C. Chen, L. Qiu, G. Xue, H. Zhu, N. Zhang, C. He, L. Pan, and C. He. An in-depth analysis of 3g traffic and performance. In *Workshop on All Things Cellular: Operations, Applications and Challenges (AllThingsCellular)*, 2015.

[63] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[64] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of LTE: Effect of network protocol and application behavior on performance. *ACM SIGCOMM Computer Communication Review*, 43(4):363–374, 2013.

[65] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[66] Y. Huang, S. Chakraborty, and Y. Wang. Using offline bitstream analysis for power-aware video decoding in portable devices. In *International Conference on Multimedia (MM)*, 2005.

[67] Y. Huang, S. Chakraborty, and Y. Wang. Watermarking video clips with workload information for dvs. In *International Conference on VLSI Design (VLSID)*, 2008.

[68] HUAWEI Technologies Co. Huawei Mate 10 Pro. https://consumer.huawei.com/us/phones/mate10/, 2017.

[69] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *International Symposium on Computer Architecture (ISCA)*, 2001.

[70] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *International Symposium on Microarchitecture (MICRO)*, 2001.

[71] Institut für Demoskopie Allensbach. Allensbacher Computer- und Technik-Analyse (ACTA) 2015, 2015.

[72] A. Iranfar, M. Zapater, and D. Atienza. Work-in-progress: a machine learning-based approach for power and thermal management of next-generation video coding on mpsocs. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2017.

[73] A. Iranli, W. Lee, and M. Pedram. Backlight dimming in power-aware mobile displays. In *Annual Design Automation Conference*, pages 604–607, July 2006.

[74] A. Iranli, W. Lee, and M. Pedram. Hvs-aware dynamic backlight scaling in tft-lcds. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(10):1103–1116, Oct 2006.

[75] JetBrains. Kotlin. `https://kotlinlang.org/`, 2018.

[76] X. Jin and S. Goto. Hilbert transform-based workload prediction and dynamic frequency scaling for power-efficient video encoding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(5):649–661, 2012.

[77] M. U. K. Khan, M. Shafique, and J. Henkel. Power-efficient workload balancing for video applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, 24(6):2089–2102, 2016.

[78] D. Kim, N. Jung, and H. Cha. Content-centric display energy management for mobile devices. In *Annual Design Automation Conference (DAC)*, 2014.

[79] 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.

[80] C. Klotzbach. Enter the matrix: App retention and engagement. *Yahoo Flurry Analytics*, 2016.

[81] A. Knox and P. Seeling. Mobile web page characteristics: Delivery and stability considerations. In *Consumer Communications Networking Conference (CCNC)*, 2017.

[82] A. Kucheria. Energy-Aware Scheduling (EAS) Project. `https://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-project/`, 2015.

[83] P. Kulkarni and P. Jaini. Android phone performance enhancement by energy efficient web browser. In *Global Conference on Communication Technologies (GCCT)*, 2015.

[84] Y.-W. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *International Conference on Software Maintenance (ICSM)*, 2013.

[85] LG Electronics. KLG V40 ThinQ. https://www.lg.com/us/mobile-phones/v40-thinq, 2018.

[86] Li, K. Xu, D. Wang, C. Peng, K. Zheng, R. Mijumbi, and Q. Xiao. A longitudinal measurement study of tcp performance and behavior in 3g/4g networks over high speed rails. *IEEE/ACM Transactions on Networking (TON)*, 25(4):2195–2208, 2017.

[87] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond. Automated energy optimization of http requests for mobile applications. In *International Conference on Software Engineering (ICSE)*, 2016.

[88] D. Li, A. H. Tran, and W. G. J. Halfond. Making web applications more energy efficient for oled smartphones. In *International Conference on Software Engineering (ICSE)*, 2014.

[89] J. Li, J. Xiao, H. Azzouz, J. W. Hong, and R. Boutaba. Powerguide: Accurate wi-fi power estimator for smartphones. In *Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2014.

[90] Y. Li, H. Deng, J. Li, C. Peng, and S. Lu. Instability in distributed mobility management: Revisiting configuration management in 3g/4g mobile networks. *SIGMETRICS Perform. Eval. Rev.*, 44(1):261–272, 2016.

[91] 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 (EMC)*, 2010.

[92] A. Limited. big.little technology. https://developer.arm.com/technologies/big-little, 2018.

[93] B. Lin, A. Mallik, P. Dinda, G. Memik, and R. Dick. User- and process-driven dynamic voltage and frequency scaling. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[94] C.-H. Lin, C.-K. Kang, and P.-C. Hsiu. Catch your attention: Quality-retaining power saving on mobile oled displays. In *Annual Design Automation Conference (DAC)*, 2014.

[95] W. Lin and C. C. Jay Kuo. Perceptual visual quality metrics: A survey. *Journal of Visual Communication and Image Representation*, 22(4):297–312, 2011.

[96] J. R. Lorch and A. J. Smith. Pace: a new approach to dynamic voltage scaling. *IEEE Transactions on Computers (TOCS)*, 53(7):856–869, 2004.

[97] M. Meeker. KPCB Internet Trends. http://www.kpcb.com/blog/2014-internet-trends, 2014.

[98] M. Meeker. KPCB Internet Trends. http://www.kpcb.com/blog/2015-internet-trends, 2015.

[99] X. Ma, Z. Deng, M. Dong, and L. Zhong. Characterizing the Performance and Power Consumption of 3D Mobile Games. *Computer*, 46(4):76–82, 2013.

[100] A. Mallik, B. Lin, G. Memik, P. Dinda, and R. P. Dick. User-driven frequency scaling. *IEEE Computer Architecture Letters*, 5(2):16–16, 2006.

[101] J. Manweiler and R. R. Choudhury. Avoiding the rush hours: Wifi energy management via traffic isolation. *IEEE Transactions on Mobile Computing (TMC)*, 11:739–752, 2011.

[102] 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.

[103] J. Min and H. Cha. Reducing display power in dvs-enabled handheld systems. In *International symposium on Low power electronics and design (ISLPED)*, 2007.

[104] G. Nam and K. Park. Analyzing the effectiveness of content delivery network interconnection of 3g cellular traffic. In *Proceedings of The Ninth International Conference on Future Internet Technologies (CFI)*, 2014.

[105] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *International Conference on World Wide Web (WWW)*, WWW, 2016.

[106] NetMarketShare. Browser Market Share. https://netmarketshare.com, 2018.

[107] B. Nguyen, A. Banerjee, V. Gopalakrishnan, S. Kasera, S. Lee, A. Shaikh, and J. Van der Merwe. Towards understanding tcp performance on LTE/EPC mobile networks. In *Workshop on All Things Cellular: Operations, Applications, Challenges (AllThingsCellular)*, 2014.

[108] J. Nielsen. *Usability Engineerings*. Morgan Kaufmann, 1993.

[109] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. Energy and performance of smartphone radio bundling in outdoor environments. In *International Conference on World Wide Web (WWW)*, 2015.

[110] S. Nirjon, A. Nicoara, C. Hsu, J. Singh, and J. Stankovic. Multinets: Policy oriented real-time switching of wireless interfaces on mobile devices. In *Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

[111] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen. Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling. In *Workshop on Power-Aware Computing and Systems (HotPower)*, 2014.

[112] W. Oliveira, R. Oliveira, and F. Castor. A study on the energy consumption of Android app development approaches. In *International Conference on Mining Software Repositories (MSR)*, 2017.

[113] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Linux Symposium*, 2006.

[114] Pandaboard.org. Omap4460 pandaboard es system reference manual.

[115] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *European Conference on Computer Systems (EuroSys)*, 2012.

[116] 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.

[117] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU power management for 3D mobile games. In *Annual Design Automation Conference (DAC)*, 2014.

[118] A. Pathania, S. Pagani, M. Shafique, and J. Henkel. Power management for mobile games on asymmetric multi-cores. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[119] U. Paul, A. P. Subramanian, M. M. Buddhikot, and S. R. Das. Understanding traffic dynamics in cellular data networks. In *International Conference on Computer Communications (INFOCOM)*, 2011.

[120] I. Pefkianakis, J. Chandrashekar, and H. Lundgren. User-driven idle energy save for 802.11x mobile devices. In *International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2014.

[121] T. Pering, Y. Agarwal, R. Gupta, and R. Want. Coolspots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2006.

[122] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 1998.

[123] N. Peters, D. Füß, S. Park, and S. Chakraborty. Frame-based and thread-based power management for mobile games on HMP platforms. In *International Conference on Computer Design (ICCD)*, 2016.

[124] 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.

[125] N. Peters, S. Park, D. Clifford, S. Kyostila, R. McIlroy, B. Meurer, H. Payer, and S. Chakraborty. API for power-aware application design on mobile systems. In *International Conference on Mobile Software Engineering and Systems (MobileSoft)*, 2018.

[126] N. Peters, S. Park, D. Clifford, S. Kyostila, R. McIlroy, B. Meurer, H. Payer, and S. Chakraborty. Phase-aware web browser power management on HMP platforms. In *International Conference on Supercomputing (ICS)*, 2018.

[127] A. Pilon. Smartphone battery survey: Battery life considered important. `https://aytm.com/blogmarket-pulse-research/smartphone-battery-survey/`, 2016.

[128] G. Pinto and F. Castor. Energy efficiency: A new concern for application software developers. *Communications of the ACM*, 60(12):68 – 75, 2017.

[129] B. Pirker. The impact of adblocking on energy consumption of mobile web browsers. Master's thesis, Technical University of Munich, 2018.

[130] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *International Conference on Mobile Computing and Networking MobiCom)*, 2001.

[131] Qualcomm Technologies, Inc. Qualcomm® Snapdragon™ 808. `https://www.qualcomm.com/products/snapdragon/processors/808`, 2015.

[132] Qualcomm Technologies, Inc. Snapdragon 845 Mobile Platform. `https://www.qualcomm.com/products/snapdragon/processors/845`, 2018.

[133] A. Rahmati and L. Zhong. Context-for-wireless: Context-sensitive energy-efficient wireless data transfer. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2007.

[134] B. Raman and S. Chakraborty. Application-specific workload shaping in multimedia-enabled personal mobile devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):10:1–10:22, 2008.

[135] K. Rasmussen, A. Wilson, and A. Hindle. Green mining: Energy consumption of advertisement blocking methods. In *International Workshop on Green and Sustainable Software (GREENS)*, 2014.

[136] E. Rattagan. Wi-fi usage monitoring and power management policy for smartphone background applications. In *Management and Innovation Technology International Conference (MITicon)*, 2016.

[137] A. Sampson, C. Cascaval, L. Ceze, P. Montesinos, and D. Suarez Gracia. Automatic discovery of performance and energy pitfalls in HTML and CSS. In *International Symposium on Workload Characterization (IISWC)*, 2012.

[138] Samsung Electronics Co., Ltd. Exynos 7 Octa (7420). `https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-octa-7420/`, 2015.

[139] Samsung Electronics Co., Ltd. Samsung Galaxy S6. `http://www.samsung.com/global/galaxy/galaxystory/s6-inside-stories/hardware/`, 2015.

[140] Samsung Electronics Co., Ltd. Exynos 9 Series (8895). `https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-8895/`, 2017.

[141] Samsung Electronics Co., Ltd. Samsung Galaxy S8. `https://www.samsung.com/global/galaxy/galaxy-s8/`, 2017.

[142] Samsung Electronics Co., Ltd. Exynos 9 Series (9810). `https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9810/`, 2018.

[143] Samsung Electronics Co., Ltd. Samsung Galaxy S9. `https://www.samsung.com/global/galaxy/galaxy-s9/`, 2018.

[144] B. Shebaro, D. Jin, and E. Bertino. Poster: Performance signatures of mobile phone browsers. In *SIGSAC conference on Computer communications security (CCS)*, 2013.

[145] H. Shim, N. Chang, and M. Pedram. A backlight power management framework for battery-operated multimedia systems. *IEEE Design and Test*, 21(5):388–396, 2004.

[146] D. Shin, Y. Kim, N. Chang, and M. Pedram. Dynamic voltage scaling of oled displays. In *Annual Design Automation Conference (DAC)*, 2011.

[147] D. Shin, Y. Kim, N. Chang, and M. Pedram. Dynamic driver supply voltage scaling for organic light emitting diode displays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1017–1030, 2013.

[148] R. Simons and A. Pras. The hidden energy cost of web advertising. *CTIT Technical Reports*, 2010.

[149] S. W. Smith. *Digital Signal Processing*. California Technical Publishing, 1999.

[150] P. Stoica and R. Moses. *Spectral analysis of signals*. Prentice Hall, 2005.

[151] S. Sultan, F. Ahsan, N. Khan, and A. M. Khan. Power aware decoding of a scalable video bit-stream. In *International Symposium on Performance Evaluation of Computer Telecommunication Systems (SPECTS)*, 2009.

[152] K. W. Tan, T. Okoshi, A. Misra, and R. K. Balan. Focus: A usable and effective approach to oled display power management. In *International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2013.

[153] The Linux Foundation. CPU boost. `https://android.googlesource.com/kernel/msm/+/android-msm-dory-3.10-kitkat-wear/drivers/cpufreq/cpu-boost.c`, 2014.

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

[155] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: Characterizing mobile advertising. In *Internet Measurement Conference (IMC)*, 2012.

[156] A. Visser. The effect of ad blockers on the energy consumption of mobile web browsing. In *Twente Student Conference on IT*, 2016.

[157] T. K. Wee and R. K. Balan. Adaptive display power management for oled displays. In *International Workshop on Mobile Gaming (MobileGames)*, 2012.

[158] T. K. Wee, T. Okoshi, A. Misra, and R. K. Balan. Focus: A demo on usable and effective approach to oled display power management. *ACM SIGMOBILE Mobile Computing and Communications Review (HotMobile)*, 17(3), 2013.

[159] A. Wei. 10 nm process rollout marching right along. `http://www.techinsights.com/about-techinsights/overview/blog/10nm-rollout-marching-right-along/`, 2017.

[160] P. Wellner, M. Flynn, S. Tucker, and S. Whittaker. A meeting browser evaluation test. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, 2005.

[161] X. Xie, X. Zhang, and S. Zhu. Accelerating mobile web loading using cellular link information. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[162] Y. Yan, S. He, Y. Liu, and L. Huang. Optimizing power consumption of mobile games. In *7th Workshop on Power-Aware Computing and Systems (HotPower)*, 2015.

[163] YouTube. YouTube for Press. `https://www.youtube.com/intl/en-GB/yt/about/press/`, 2018.

[164] 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 (TMC)*, 14(8):1698–1711, 2015.

[165] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *SIGOPS Operating Systems Review (OSR)*, 37(5):149–163, 2003.

[166] W. Yuan and K. Nahrstedt. Energy-efficient cpu scheduling for multimedia applications. *ACM Transactions on Computer Systems (TOCS)*, 24:292–331, 2006.

[167] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.

[168] B. Zhao, W. Hu, Q. Zheng, and G. Cao. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26(3):761–774, 2015.

[169] Y. Zhu, M. Halpern, and V. Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[170] Y. Zhu, M. Halpern, and V. J. Reddi. The role of the cpu in energy-efficient mobile web browsing. *IEEE Micro*, 35(1):26–33, 2015.

[171] Y. Zhu and V. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[172] 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):33–36, 2014.

[173] L. Zou, A. Javed, and G. Muntean. Smart mobile device power consumption measurement for video streaming in wireless environments: Wifi vs. lte. In *International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, 2017.