Web Browser Workload Characterization for Power Management on HMP Platforms

Nadja Peters[1], Sangyoung Park[1], Samarjit Chakraborty[1],
Benedikt Meurer[2], Hannes Payer[2], Daniel Clifford[2]
Technical University of Munich[1], Google Inc[2]
{nadja.peters, sangyoung.park, samarjit}@tum.de, {bmeurer, hpayer, danno}@google.com

ABSTRACT

The volume of mobile web browsing traffic has significantly increased as well as the complexity of the mobile websites 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 a heterogeneous multi-processing (HMP) platform found in many smartphones. 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 propose power management strategies based on the results. All trace data and measurement results have been collected on a real HMP platform integrating the Samsung Exynos5422 SoC, also used in the Samsung Galaxy S5 smartphone. Our work shows that there is a considerable scope for power savings and outlines directions for future research. We believe that it will lead to development of practical power management techniques considering thread allocation, dynamic voltage and frequency scaling (DVFS) and power gating.

Keywords

Power Management; DVFS; Heterogeneous Multi-Processing; big.LITTLE; Mobile Web Browser; JavaScript Engine

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 major ten countries including US and Japan [8]. This trend is likely to accelerate with the exploding sales of tablet devices which has grown ever faster than PCs [12]. Not only the mobile web traffic, but also computation demand of the mobile web pages is significantly increasing [17].

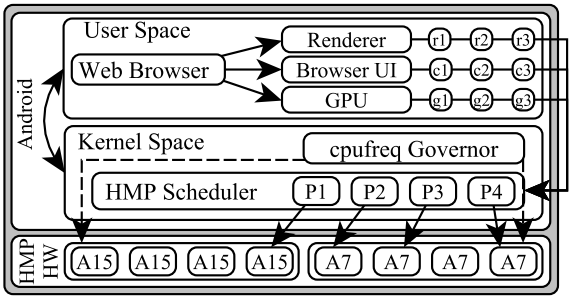

Figure 1: Web Browser with processes and threads running on an HMP platform.

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 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 [13]. 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 [6]. 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.

Hence, existing power management techniques for web browsing workload 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 a time being, and the scheduler allocates and schedules tasks based on CPU usage thresholds not specifically taking into consideration the

CODES/ISSS '16 October 01-07 2016, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-4483-8/16/10.
DOI: http://dx.doi.org/10.1145/2968456.2968469

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 1. This architecture is adopted in state-of-the-art smartphones like the Nexus 5X with its Qualcomm Snapdragon 808 processor [7, 14] and the Samsung Galaxy S6 with its Exynos Octa 7420 [16, 15]. 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 paper provides a non-trivial, detailed analysis of the actual thread workload 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 paper 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 Hypertext Markup Language (HTML), Cascading Style Sheets (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 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 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 and suggest potential architectures.

- 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 work.

The rest of the paper is organized as follows: Section 2 gives an overview of related work in the area of browser power management. In Section 3, we give details on browser internals and web browser application characteristics. Then, we discuss drawbacks of the Android power management according to web browsing on the HMP platform in Section 4. We introduce our hardware and software measurement infrastructure in Section 5. In Section 6, 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 7. We discuss potentials for future work in Section 8.

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 has been works putting emphasis on the network power consumption during web browsing. In [19], it has been 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 reduces the power consumption by grouping the data transmissions during page loading and letting the 3G radio interface sleep more [4]. However, recent work reveals 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 website rendering power consumption show that downloading and parsing CSS

as well as JavaScript consumes a significant amount, up to 50%, of total power [13].

There has been some research that characterize 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 [1]. It focuses on showing up energy pitfalls for web page design. In [13], the authors propose power saving techniques based on web page modification and browser computation offloading to a remote proxy. However, they do 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 has been an advanced line of work on this topic that studies web browsing power consumption on HMP platforms. In [17], a predictive model based on web page primitives is introduced. This model is used to find the appropriate core and operating frequency according to web pages in a heterogeneous system. However, they use a setup that consists of two separate platforms incorporating a big and a little CPU to validate their approach, which poses as a weakness. Another work identifies quality of service (QoS) requirements of different mobile web applications by event-profiling to perform DVFS on a big.LITTLE platform [18]. Again, they do not observe actual thread-level workloads of web browsers. We take a cue from these studies and analyze the different processes and their threads with the JavaScript engine to evaluate their power consumption.

In summary, the main contribution of this work 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 paper - e.g., determining power-aware thread allocation schemes to CPU cores. Moreover, to the best of our knowledge, we are the first to show up power saving potentials and propose power management techniques for web browsers on an HMP platform by exploiting all available mechanisms on this platform, such as power gating, DVFS and HMP scheduling.

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.

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 layout. Scripting languages like PHP and JavaScript are used for dynamic and interactive elements such as user inputs or slide shows. Among the scripting languages, we focus on JavaScript as research has already shown its large impact on web browser power consumption [13]. JavaScript intensive phases occur during the loading of a page or user interaction.

3.2 Components of a Browser

The main components of a browser are the browser engine, rendering engine and JavaScript engine as shown in Figure 2 [5]. The browser engine acts as 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. 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.

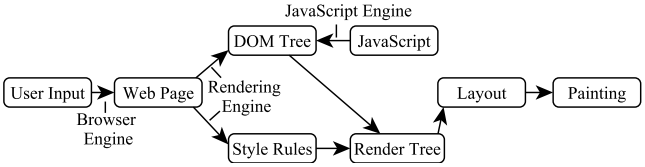

Figure 2: Schematic structure of a browser.

3.3 JavaScript Engine V8

As our experiments were performed using the Chrome browser, we focus on describing the internals of V8 [6], 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. 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.

3.4 Web Page Rendering

The browsing process can be separated into a loading phase and a post loading phase as depicted in Figure 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 [13]. 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 [2], a company 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. HETEROGENEOUS MPSOC ARCHITECTURE AND POWER MANAGEMENT

In this section, we describe the HMP hardware platform used in this work, how the Android operating system power management works for this kind of architecture and why the current Android power management strategy does not perform optimally in terms of power consumption.

4.1 Hardware Platform

The underlying hardware platform that we evaluate is the Odroid-XU3 board which features an Exynos5422 SoC also used in the Samsung Galaxy S5 phone [10]. As discussed before, the heterogeneous MPSoC is based on ARM big.LITTLE architecture which incorporates two different CPU clusters. One is a power-optimized Cortex-A7 quad-core CPU (A7) and the other is a performance-optimized Cortex-A15 quad-core CPU (A15). In the following, we refer to a CPU cluster as *CPU* while we refer to single CPU cores of a CPU cluster as *core*. CPU voltage and frequency values can be adjusted independently per CPU but not per core. The A7 supports a frequency range from 1.0 to 1.4 GHz and the A15 from 1.2 to 2.0 GHz. The platform also supports power gating, but it is only available at CPU cluster granularity. The platform runs an Android Kitkat 4.4.4 with a Linux kernel version 3.10.9. The setup features HMP scheduling which allows to distribute tasks over all big and little cores at the same time. The HMP scheduler prefers the small cores first, and then migrates threads to the big cores if the CPU utilization goes above a certain threshold.

4.2 Power Management on HMP Platforms

The power management in Android is handled by three individual components, mainly residing in the Linux kernel: (1) the *scheduler*, (2) the frequency *governor* and (3) the *wakelock* mechanism. The three of them work independently of each other as depicted in Figure 3.

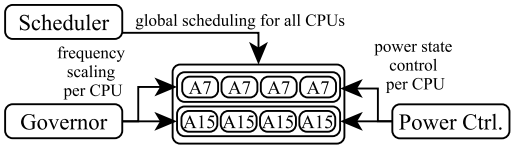

Figure 3: Traditional Android power management with independent control units.

The scheduler distributes tasks based on their priorities, allocating them to cores while the governor is a part of the *cpufreq* driver regulating the CPU frequency. The Android Open Source Project (AOSP) by Google contains different governors of which the most widely used are the *ondemand* and the *interactive* governors. They both periodically monitor the CPU load and adaptively adjust the operating frequency. Note that different CPUs can have different governors. When the CPU load exceeds a certain threshold (in our setup 95%), *ondemand* governor ramps up to the maximum frequency. The *interactive* governor is developed for mobile devices and designed to react to user inputs faster. Whenever the CPU wakes up from idle mode, the CPU load is monitored to ramp up the frequency immediately if necessary. The wakelock mechanism which is a part of Android and not the Linux kernel keeps the device in a wake-state as long as an application is executing. When the device becomes idle, the processor is put in a power-saving sleep state as soon as possible. However, the A15 is not power gated when it is not utilized, but the device is in use. Moreover, state-of-the-art power-aware scheduling techniques such as consolidating the workload on one A15 core and shutting down the other cores are not reflected by the Android default governors. This is due to the above mentioned impracticality of turning off separate A15 cores, because of which this strategy is not applicable for this platform.

Limits of Android Power Management: The division described above works well for traditional hardware architectures with 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 heterogeneous multiprocessing scheduler is offered by Samsung [9]. 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. An example for this situation is shown in Figure 4. It shows the loading of the ebay web page on the target platform using the regular Android settings.

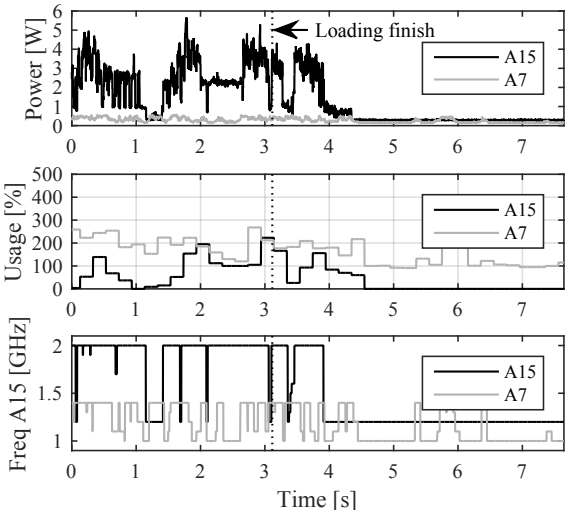

Figure 4: Loading eBay web page (regular Android settings).

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 phase generates a high workload, and consequently, the A15 was enabled. Although neither of the CPUs are fully loaded, one

can see that the corresponding clock frequencies are switched between the minimum 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. Therefore, these results show a considerable scope for power savings. (3) In the post loading phase, 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. 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 rather power gated. This is possible on a CPU cluster basis for the A15.

We believe that an approach that combines the three power management components can lead to significant power savings. For example, the scheduler could turn off the A15 since it knows which cores all threads are scheduled on. Therefore, we study the power saving potential of these components for web browsing in a the following chapters. Based on these observations and an analysis of the web browser workload, we propose potential power management techniques for web browsing on HMP platforms.

5. MEASUREMENT SETUP

We have implemented a software measurement framework on top of the commercially available Odroid-XU3 hardware platform. 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.

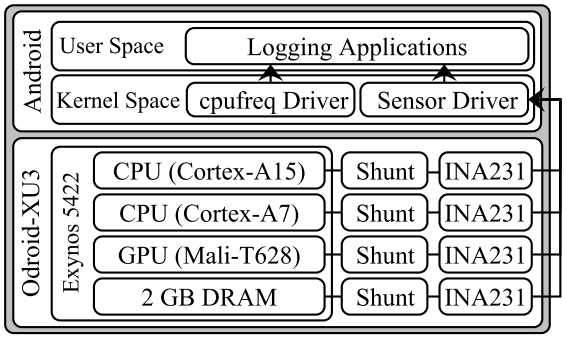

Figure 5: Exynos5422-based measurement setup.

5.1 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 5. 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.

5.2 Software Infrastructure

Our software setup is a combination of three different logging environments as shown in Figure 6, (1) the power logger, (2) the process logger and (3) the Chrome:trace environment. Combining all these information, we are able to

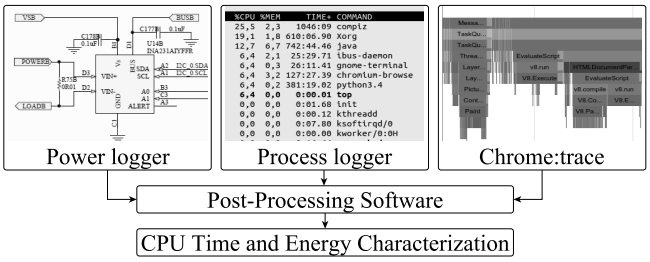

Figure 6: Software setup for data acquirement.

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 data to identify the effects of DVFS and thread allocation on the performance of the browser.

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 core allocation for traditional chips including only one CPU is not relevant to extract power information per thread, 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.

6. 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 workload 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.

6.1 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.

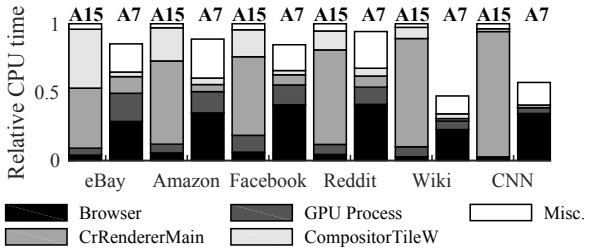

Figure 7: Relative CPU time of web browser threads for representative websites per A15 and A7 cluster.

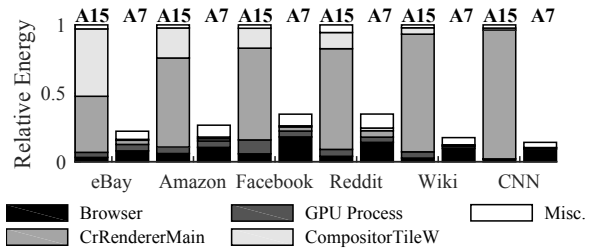

Figure 8: CPU energy consumption of threads per A15 and A7 cluster for representative websites.

Figure 7 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 8. 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 7 that different web pages exhibit different degree 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 7.2, 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. 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 8 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 7.2.

6.2 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 9 depicts the energy consumption of the rendering process divided by the main web page components CSS, HTML and V8 described in Section 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.

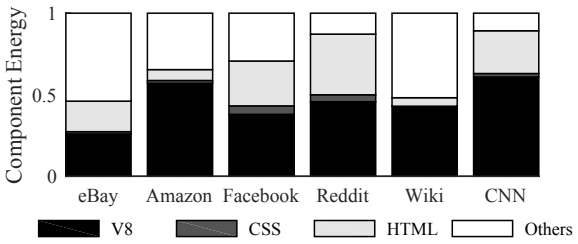

Figure 9: Relative energy distribution of the web page components HTML, CSS and JavaScript.

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 3.3 and studied the distribution of V8 workload over threads within the rendering process.

Figure 10 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.

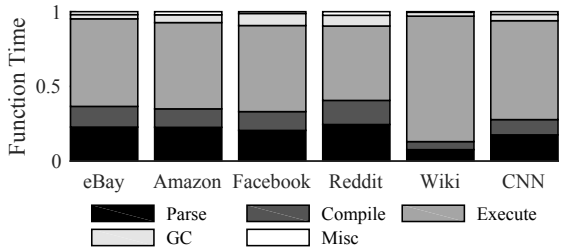

Figure 10: Relative time distribution of V8-related function calls by category.

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. These information can be exploited for power saving by thread allocation, for example moving the ScriptStreamerThread to the A7 considering its penalty on performance.

7. 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 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 6, we show a non-intuitive power-aware thread-to-core allocation strategy in Section 7.2 and outline the potential for energy savings in Sections 7.3 and 7.4.

7.1 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 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.

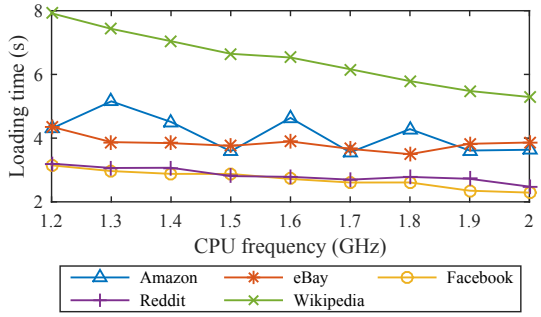

Figure 11: Web page loading time according to different A15 frequency cap.

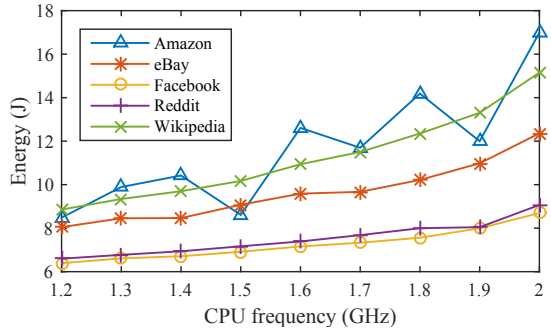

Figure 12: A15 energy consumption according to different A15 frequency cap.

Figure 11 and 12 show the loading time and energy consumption for the test scenario described in Section 6 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.

7.2 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 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 7, 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.

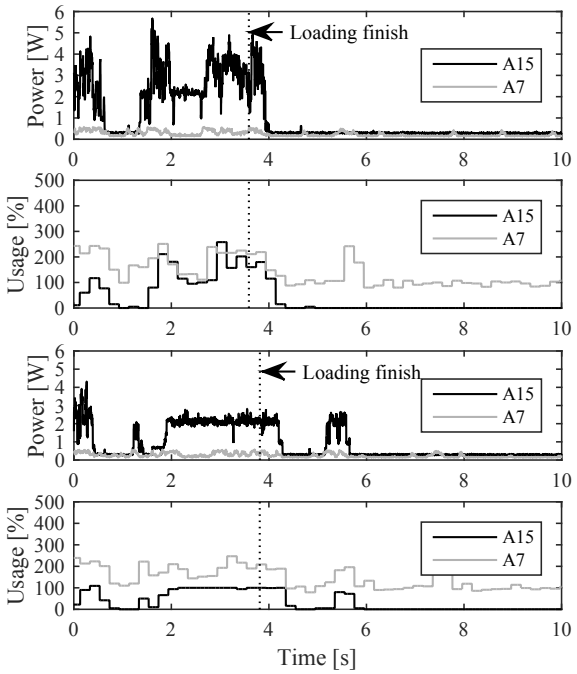

Figure 13: Power consumption and CPU utilization for loading eBay for case 1 (top, energy: 13.31 J) and case 2 (bottom, energy: 11.57 J).

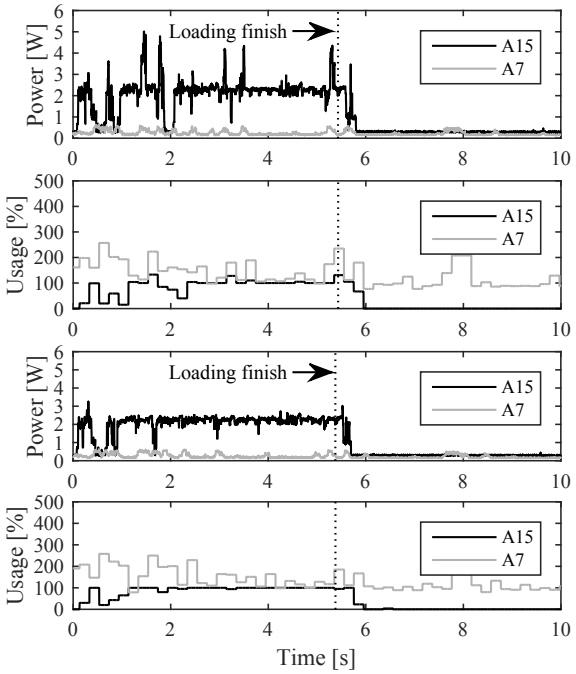

Figure 14: Power consumption and CPU utilization for loading Wikipedia for case 1 (top, energy: 15.40 J) and case 2 (bottom, energy: 15.26 J).

Comparison of case 1 and case 2 for eBay: Figure 13 shows the eBay loading phase for case 1 (top) and for case 2 (bottom). Obviously, the A15 CPU utilization goes up to 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 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 14. In contrast to eBay, less degree of thread parallelism exists in 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 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 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 15. The power consumption of the A7 is nearly the double of the cases 1 and 2, but its absolute value is significantly smaller compared with 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 1. As we have described in 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.

7.3 Power Savings by DVFS without Performance Compromise

As discussed in Section 4.2, default governors for Android often fail to assign energy-optimal frequency 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 16 shows the estimates of power consumption, CPU utilization, and operating fre-

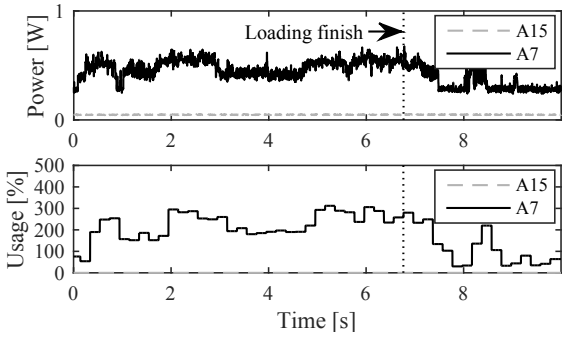

Figure 15: Power consumption and CPU utilization for loading eBay for case 3.

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

quency 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 16 is obtained by using the following CPU power model

$$P_{cpu} = u \cdot C_{eff} \cdot V^2 f + P_{static}(V), \quad (1)$$

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%.

7.4 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 applies 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 $P_{idle} = 0.27$ W, while it is only $P_{off} = 0.04$ 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.

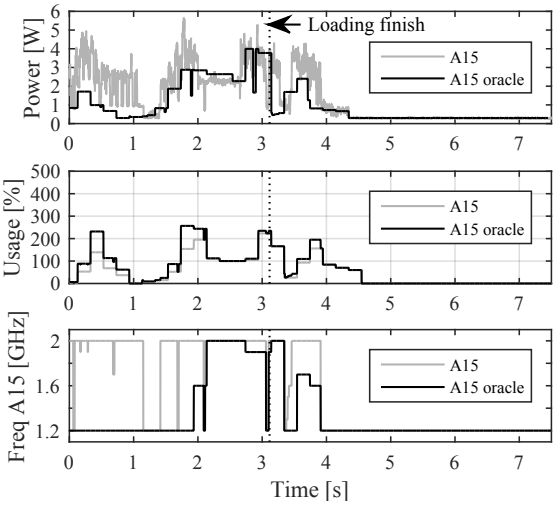

Figure 16: Frequency, usage and power of the A15 estimated by the oracle verses a real measurement when loading eBay.

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 6 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 workload. However, a more elaborate power gating technique requires detailed knowledge of the time and power overhead, which we leave as a future work.

8. FUTURE WORK

In addition to the power management possibilities outlined in the previous sections, our results show that there is a considerable potential for future work. In this section, we discuss a browsing phase-aware approach as well as the integration of the Android power management components for HMP platforms. All techniques can be applied in parallel.

Phase-aware Frame Rate Adaptive Power Management: In Section 3.4, we introduce two phases during web browsing, the loading phase and the post loading phase. Phase-aware power management in mobile web browsers has not been extensively studied, but it has been proven to be

useful for other domains of applications such as games [3]. Mobile web page rendering, like interactive games, also consists of multiple phases that vary in user requirement and user interaction such that significant power savings could be achieved using a similar approach. In our work, we have shown that the workload highly varies depending on the phase and we expect that the user requirement such as frame rate will also vary. The main objective during the loading phase is to display the web page as fast as possible. However, the strategy of the web browser is to re-render the loading web page often to illustrate an on-going loading progress. This can result in a high computation overhead due to fast web page updates and an unnecessary high frame rate. Meanwhile, that computation power could be used to process background information such as CSS and JavaScript, resulting in a faster page setup and, hence, lower loading time. During the post loading phase, the frame rate and the workload highly depend on the type of web page and its degree of interactivity. While static web pages do not need a frame rate update at all, the frame rate for dynamic contents such as slide shows or scrolling movements may vary. By adaptively changing the frame rate, we can also adapt the CPU frequency to compute a frame within a desired time bound. Finding out the frame rate requirements of different web pages and automatically adapting the frame rate will be a major part of our future work.

Integrated Android Power Manager: In Section 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. Even though 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* [11] for big.LITTLE systems is an ongoing project pushed by ARM and Lenaro, an integrated power manager capable of actually performing 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. The integrated power manager will 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.

9. CONCLUDING REMARKS

This paper provides a detailed look into the web browser workload on HMP platforms and seeks potential power savings based on the observations. Unlike previous works that analyze the power consumption according to the inputs to the web browser, the new aspect of our 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. Based on the characterization, we apply several power management techniques, such as DVFS, thread allocation to CPU cores and power gating. Moreover, we outline the theoretical power saving potential for web browsing in Android. 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.

Acknowledgments: This work was partially supported by Google Inc and by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology as part of the EEBatt project.

10. REFERENCES

[1] A. Sampson et al. Automatic discovery of performance and energy pitfalls in html and css. In *IISWC*, 2012.

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

[3] B. Dietrich et al. Forget the battery, let's play games! In *ESTIMedia*, 2014.

[4] B. Zhao et al. Energy-aware web browsing on smartphones. *TPDS*, 26(3), 2015.

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

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

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

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

[9] H. Chung et al. Heterogeneous multi-processing solution of Exynos 5 Octa with ARM® big.LITTLE technology. In *Samsung White Paper*, 2012.

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

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

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

[13] N. Thiagarajan et al. Who killed my battery: Analyzing mobile browser energy consumption. In *WWW*, 2012.

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

[15] Samsung Electronics Co., Ltd. Exynos Octa 7420. http://www.samsung.com/semiconductor/minisite/Exynos/w/solution.html#?v=7octa_7420, 2015.

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

[17] Y. Zhu et al. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.

[18] Y. Zhu et al. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *HPCA*, 2015.

[19] Y. Zhu et al. The role of the cpu in energy-efficient mobile web browsing. *IEEE Micro*, 35(1), 2015.