Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading

Duc Hoang Bui†, Yunxin Liu*, Hyosu Kim†, Insik Shin†, Feng Zhao*
†KAIST, Daejeon, South Korea, *Microsoft Research, Beijing, China
{ducbuihoang, hyosu.kim, ishin}@kaist.ac.kr, {yunliu, zhao}@microsoft.com

ABSTRACT

Web browsing is a key application on mobile devices. However, mobile browsers are largely optimized for performance, imposing a significant burden on power-hungry mobile devices. In this work, we aim to reduce the energy consumed to load web pages on smartphones, preferably without increasing page load time and compromising user experience. To this end, we first study the internals of web page loading on smartphones and identify its energy-inefficient behaviors. Based on our findings, we then derive general design principles for energy-efficient web page loading, and apply these principles to the open-source Chromium browser and implement our techniques on commercial smartphones. Experimental results show that our techniques are able to achieve a 24.4% average system energy saving for Chromium on a latest-generation big.LITTLE smartphone using WiFi (a 22.5% saving when using 3G), while not increasing average page load time. We also show that our proposed techniques can bring a 10.5% system energy saving on average with a small 1.69% increase in page load time for mobile Firefox web browser. User study results indicate that such a small increase in page load time is hardly perceivable.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications—*Information Browser*

General Terms

Design; Measurement; Performance

Keywords

Smartphones; Mobile Web Browser; Web Page Loading; Energy Efficiency

1. INTRODUCTION

Web browsing is one of the core applications on smartphones and other mobile devices such as tablets. However, web browsing, particularly web page loading, is of high energy consumption. As energy is a paramount concern on smartphones, it is desirable to improve the energy efficiency of web browsing, particularly web page loading. In this paper, we seek to reduce the energy consumption of web page loading on smartphones without compromising user experience. In particular, we aim to not increase page load time.

To achieve this goal, we study browser internals and system behaviors to understand how the energy is spent in loading web pages, and to identify opportunities to improve the energy efficiency. Although many browser manufacturers have made an effort on improving energy efficiency for mobile devices, our findings indicate that the current mobile browsers are not yet fully energy optimized for web page loading. First, the web resource processing is aggressively conducted regardless of network conditions at the risk of energy inefficiency. Second, the content painting rate is unnecessarily high, consuming a lot energy without bringing user-perceivable benefits. Finally, the power-saving capability of modern CPUs with the emerging ARM big.LITTLE architecture [3] is under-utilized. Fundamentally, the web page loading is overly optimized for performance but not for energy cost.

We argue that the energy-performance trade off must be reconsidered for web page loading on mobile devices. Based on our findings, we formulate new design principles for energy-efficient web page loading. Based on these principles, we develop three new techniques, each one addressing one of the above energy-inefficiency issues. First, we propose to use the *network-aware resource processing* (NRP) technique to effectively trade off performance for energy reduction adapting to changing network conditions. We use adaptive resource buffering to control the speed of web resource processing dynamically with regard to the speed of resource download, in order to become energy efficient without increasing page load time. Second, we propose the *adaptive content painting* (ACP) technique to avoid unnecessary content paints to reduce the energy overhead. We study the trade off between energy saving and page load time increase to ensure the user experience is not compromised. Finally, to better leverage the big.LITTLE architecture, we propose the *application-assisted scheduling* (AAS) technique to leverage internal knowledge of the browser to make better scheduling decisions. Specifically, we employ adaptive thread scheduling based on QoS feedback in a way that the browser keeps threads running on *little* cores to save energy, as long as their related QoS requirements are being satisfied.

We have implemented all the three techniques on commercial smartphones, by revising the Chromium browser. Collectively, these techniques achieve significant reductions in the energy cost of web page loading. Experimental evaluations using the Alexa top 100 websites in the U.S. [2] show that, on a big.LITTLE smartphone using WiFi, our revised Chromium browser is able to achieve 24.4%

MobiCom'15, September 7–11, 2015, Paris, France.

DOI: http://dx.doi.org/10.1145/2789168.2790103.

average system energy saving while decreasing 0.38% page load time, compared to the default Chromium browser. When using 3G, the average system energy saving is 22.5% and the average page load time increase is 0.41%. On another smartphone without big.LITTLE supported, our techniques can still reduce the energy consumption by 11.7% on average when using WiFi. We have also conducted a user study to confirm that our techniques do not affect user perceived experience. Moreover, all participants show their intention (or interest) to use our techniques either all the time (72%) or when the battery level is low (28%).

Although our implementation is based on Chromium, our proposed techniques can be also be applied to other mobile browsers. The first two energy-inefficiency issues of Chromium are related to the general procedure of web content downloading, processing, and painting. Thus, other mobile browsers are generally subject to the same issues, though with different levels of significance. Besides Chromium, we also have applied our ACP and AAS techniques to Firefox, even without a deep understanding of Firefox internals, and it resulted in a significant energy saving (10.5%) on average with a marginal increase (1.69%) in page load time, compared to the default Firefox.

The main contributions of this paper are:

1. We show that web page loading on popular mobile web browsers, including Chrome, Firefox, Opera Mobile, and UC Browser, is overly optimized for performance but not for energy-efficiency, and identify the opportunities to reduce the energy cost.
2. We derive design principles and propose techniques to improve the energy efficiency of web page loading on smartphones.
3. We implement our techniques by revising the Chromium browser, demonstrating the applicability of our techniques in real-world mobile browsers. We also implement the proposed ACP and AAS in the Firefox browser, showing that our findings can be applied to other mobile browsers.
4. We conduct comprehensive experiments and a user study to show that our techniques effectively reduce energy consumption of mobile web page loading without compromising the user experience.

The rest of this paper is organized as follows. Section 2 introduces the Chromium browser and the big.LITTLE architecture as the background. Section 3 reports the energy-inefficiency issues and derive general guidelines to reduce the energy cost. Section 4 describes our novel techniques. Section 5 shows the implementation details and Section 6 presents the evaluation results. Section 7 discusses more aspects on the energy-saving techniques and the future work, Section 8 surveys the related work, and Section 9 concludes.

2. BACKGROUND

In this section, we introduce the Chromium browser and the big.LITTLE architecture as the background.

2.1 Chromium Browser Architecture

Figure 1 shows the architecture of the Chromium web browser. Chromium uses a multi-process architecture of a single *Browser* process and multiple *Renderer* processes. Each Renderer process runs an instance of the rendering engine (previously WebKit [14] and now Blink [4]) and the JavaScript engine that parse and execute web content. Each Renderer process typically corresponds to one tab in the web browser UI. The Browser process runs the network stack and fetches web resources from the network for all Renderer processes, allowing efficient network resource sharing among

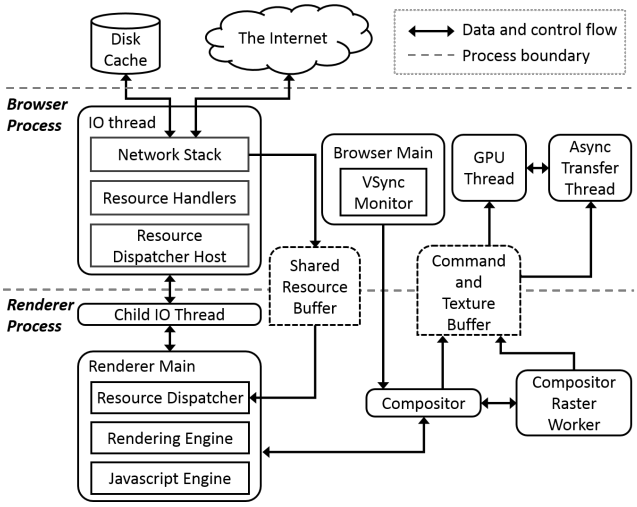


Figure 1: Architecture of the Chromium web browser

all Renderer processes. Renderer processes run in a sandboxed environment with limited access to the client device and the network, preventing exploits in rendering engine from compromising the whole web browser.

Each process in Chromium has multiple threads. Table 1 provides a brief description of the main threads. Processes and threads are designed to work asynchronously. The Browser process and a Renderer process communicate with each other through IPC mechanisms (e.g., named pipes) and exchange data using shared memory. The Browser process puts the fetched web resources into the Shared Resource Buffer from which the Renderer process reads the data to create graphic layers for the corresponding web page. The Compositor thread then saves the generated graphic data into the Command and Texture Buffer for GPU processing in the Browser process, because the sandboxed Renderer process does not have the privilege to access the GPU directly. The GPU thread generates the final screen image into the display frame buffer to be displayed on the device screen.

The graphic composition and GPU processing are driven by a callback from Android framework called vertical synchronization (VSync). Each VSync signal indicates the beginning of a display frame, so that graphic data should be generated and moved to the display frame buffer for drawing on the screen before the next VSync. VSync signals are generated at 60 times per second. In the Browser process, the VSync Monitor of the Browser Main thread monitors VSync signals and forwards them to the Compositor thread in a Renderer process.

The architecture shown in Figure 1 also applies to Google Chrome, Opera Mobile and Android stock browser which are built on top of Chromium source code [6, 19]. The browsers share the same rendering engine, JavaScript engine, and underlying core modules.

2.2 big.LITTLE Architecture and Scheduling

The ARM big.LITTLE architecture [3] combines high performance and high power *"big"* cores together with power efficient but low performance *"little"* cores in a single ARM CPU. By running lightweight tasks on the little cores and heavy tasks on big cores, a big.LITTLE CPU is able to meet both the need for higher performance and the desire for longer battery life on mobile devices. For example, the Samsung Galaxy S5 SM-G900H smartphone uses a Samsung Exynos 5422 System-on-Chip (SoC) with 4 Cortex-A15 (big) cores and 4 Cortex-A7 (little) cores. The Exynos 5422 SoC

Thread name	Description
CrBrowserMain	Starts and initializes the browser, monitor VSync callbacks, and renders the user interface
CrRendererMain	Runs the rendering engine and the Javascript engine
Chrome_IOThread	Communicates with other processes (such as Renderer and GPU processes), and the environment (such as the network and the disk cache)
Chrome_ChildIOThread*	Communicates with other processes using the IPC mechanism
Compositor	Generates platform-independent GPU commands to composite the web page layers into the final screen image
CompositorRasterWorker*	Rasterizes the web page layers into graphic textures
Chrome_InProcGpuThread	Executes GPU commands issued by the Compositor thread to draw textures of each layer into the frame buffer for displaying the final web page image on the screen
AsyncTransferThread	Uploads textures provided by Compositor to the GPU memory
SimpleCacheWorker*	Reads and writes web resources on the browser disk cache

$*$ possible multiple thread instances

Table 1: Main threads in the Chromium web browser

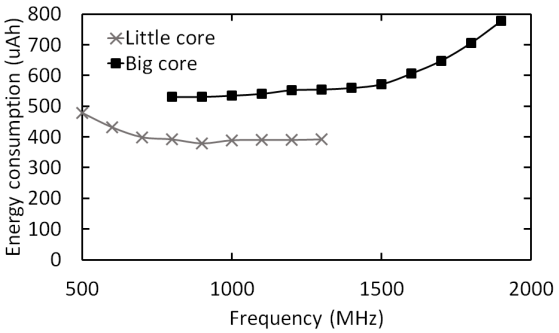


Figure 2: Total energy consumption of a little core and a big core on a Samsung S5 SM-G900H smartphone in running the same workload with a fixed number of operations

uses Heterogeneous Multi-Processing (HMP) [18], so it can utilize all the 8 cores simultaneously. The little and big cores can operate in a frequency range from 500 MHz to 1.3 GHz and from 800 MHz to 1.9 GHz, respectively. Figure 2 shows our measured total energy consumption of a little core and a big core of the Galaxy S5 smartphone in running the same workload, at different frequencies with 100% core utilization. The workload is a busy loop for a fixed number of iterations which emulates the characteristics of CPU-bound workloads with no periods of idleness. At its highest frequency of 1.3 GHz, a little core can reduce the energy consumption by a factor of 2 (392 uAh vs. 777 uAh), compared to a big core at its own highest frequency of 1.9 GHz. The measurement result is consistent with a previous study [17] and demonstrates the big potential of big.LITTLE architecture in saving energy.

In the Linux kernel, the task scheduler divides all the cores into two scheduling domains: one for the little cores and the other for the big cores [37]. The scheduler may migrate a thread from the big core cluster to the little core cluster or vice versa, by tracking the loads of all the threads. Basically, the scheduler will move a thread from a little core to a big core if the load contribution of a thread is larger than an *up threshold* and vice versa with a *down threshold*.

Commercial big.LITTLE smartphones have become increasingly popular. Beside Samsung, other mobile processor makers including Qualcomm, LG, Huawei and MTK have also shipped or announced their big.LITTLE products.

3. UNDERSTANDING ENERGY COST OF PAGE LOADING

In this section, we study Chromium browser internals and the system-scheduling behaviors and identify three energy-inefficient issues in loading web pages: inefficient web resource processing, unnecessary content painting, and underutilized little cores. We also study the behaviors of other mobile browsers and show that they also have the same energy-inefficient issues. Based on the findings, we derive general design guidelines for energy-efficient web page loading.

Unless otherwise specified, all the numbers presented in this section are the average experiment results obtained on a Samsung Galaxy S5 SM-G900H big.LITTLE smartphone, using the Alexa top 10 websites in the US [2], including Google, Facebook, Youtube, Yahoo, Amazon, Wikipedia, Linkedin, Ebay, Twitter, and Pinterest. The websites were loaded on an emulated 3G network (Download 2 Mbps, Upload 1 Mbps, RTT 120 ms) for repeatable experiments. The total system energy consumption of the smartphone was measured using a Monsoon power monitor [29].

3.1 Inefficient Web Resource Processing

Web resource processing is one of the main components of browsing, along with resource downloading and content painting. Web resource processing typically includes HTML and CSS parsing, image decoding, Javascript execution, and updating DOM tree and graphic layers. In Chromium, the Renderer process conducts the above processing whenever it is notified of the data available from the Browser process. As shown in Figure 1, those two processes exchange data using a shared resource buffer. The Browser process uses the $read$ system call on a socket handler to receive the resources of a web page from the network. Each $read$ system call directly writes received data into the resource buffer with a maximum data size of 32 KB by default. Upon each $read$ system call, the Browser process immediately notifies the Renderer process to start to process the received data, no matter how much data is received.

Although immediately processing the received data is natural and minimizes the page load time, it is not energy efficient. The reason is that many $read$ system calls return a small amount of data and thus lead to a large number of IPCs between the Browser process and the Renderer process. In loading the 10 websites, 99% of the $read$ calls return a data chunk less than 3 KB and the average data size returned per $read$ is only 1.3 KB. On average there are 319 $read$ calls (62 per second) in loading one web page. Each $read$ may cause multiple IPCs between the Browser process and the Renderer process. On average the total number of IPCs per web page is 871 (176 per second). Each IPC has a fixed overhead. Moreover, there are also other overheads in web resource processing. Each time when processing a data chunk, even if the data chunk is small, the Renderer process has to go through the whole data rendering pipeline. In particular, for image data, many graphic

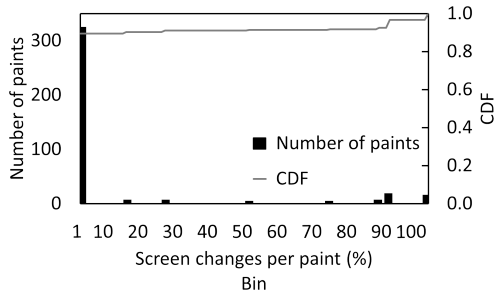


Figure 3: Visible screen update in loading instagram.com

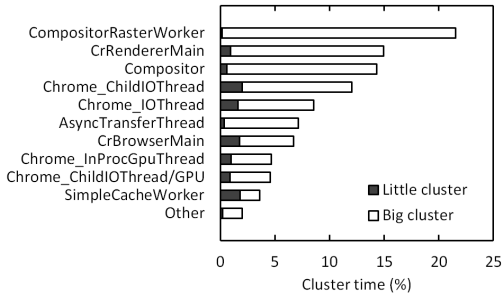


Figure 4: Execution time of Chromium threads on the big and little cores in loading instagram.com

activities are involved in the Compositor, Raster Worker and Async Transfer threads. As a result, the accumulated overheads are high and thus waste much energy, more than 10% of the total system energy cost as we will show in Section 6.

3.2 Unnecessary Content Painting

When loading a web page, Chromium processes the web resources progressively and keeps updating the partially rendered display results onto the screen. As shown in Figure 1, the screen updates are driven by VSync signals. Upon receiving a VSync signal, the Compositor thread ships the current rendered graphic data to the GPU thread of the Browser process through the shared command and texture buffer. The GPU thread then executes platform-specific graphic commands on the GPU to composite all the textures into the final graphic result to be displayed on the screen. This graphic pipeline of data rendering and screen updating is called *content painting*. Many content paints may occur in loading a web page. For example, in loading *instagram.com*, a popular photo-sharing website, the browser does 359 paints (23.1 per second) even the web page does not have any animation.

However, we have observed that most of the times, content painting generates a very small or even zero visible change on the screen. To demonstrate this observation, we log the visual progress of each content painting using Telemetry [11], the performance testing framework of Google Chrome. The tool collects the $Paint$ events from the rendering engine and calculates how many visible pixels of the screen are changed by each $Paint$ event. Figure 3 shows the results in loading *instagram.com*, including both the histogram and the CDF. Among all the 359 paints, 321 (89%) of them generate a zero visible change on the screen. For the top 10 websites, there are also many paints with a zero or very small visible screen change: 56% of the paints do not generate any visible screen change, and 62% of the paints generate a visible screen change of less than 5%. These paints with a zero or very small visible screen change do not help improve the user experience. However, they cause overheads in the whole graphic processing pipeline and the IPCs between the two processes, and thus lead to unnecessary energy cost. This design of high-rate content painting is overly optimized for performance, not energy efficient, and must be re-considered on mobile devices.

3.3 Underutilized Little Cores

Although big.LITTLE is designed to save energy, we find that the energy-saving potential of big.LITTLE is not fully exploited in the case of Chromium. Specifically, we find that the little cores are underutilized. For example, Figure 4 shows the average execution time of Chromium threads in loading *instagram.com*. Most of the times, the threads are executed on the big cores. 89% of the total execution time is on the big cores but only 11% of the total execution time is on the little cores. Some threads, such as the Compositor Raster Worker and Async Transfer thread, are almost completely (more than 95%) executed on the big cores.

The reason of low little-core utilization is that the OS schedules threads based on a load-driven approach that favors performance more than energy saving. The scheduler tries to finish a thread sooner rather than saving more energy. On symmetric multi-core architectures, since all the CPU cores are equal, this load-driven approach is energy efficient. If a thread can finish its task faster, the CPU can go to sleep sooner to save energy [16]. However, on heterogeneous multi-core architectures like big.LITTLE, finishing a thread sooner may not reduce the energy cost. As shown in Figure 2, a little core has a lower energy per instruction cost than a big core. As a result, it takes longer but consumes less energy to run a thread on a little core than on a big core. Thus, a thread should run on a little core as long as it can tolerate a resulting delay. However, the OS cannot know how much delay a thread can tolerate, and thus cannot decide whether to run the thread on a little core for better energy efficiency or not.

3.4 Guidelines for Improvements

The above energy-inefficient issues are not Chromium-specific. Even without analyzing source codes, we observe the behaviors of several other mobile browsers when they load the same set of web pages. After monitoring the number of screen updates and thread scheduling behaviors, we conclude that those mobile browsers basically share the same energy-inefficiency issues with Chromium: they also do aggressive content processing and screen updating, and rely on the OS for passive thread-scheduling.

We monitor the screen update by dumping information from Android SurfaceFlinger. For example, when loading *instagram.com*, Firefox updates its SurfaceView 614 times in 17.1 sec or 35.9 frames/s. Similar behavior is also observed on Opera Mobile web browser which is based upon Chromium. Another popular web browser, UC Browser, even keeps updating at a fixed 60 times per second even there is no change on the content. Thereby, even when the content does not change, UC Browser has 1755 mW average power consumption while other browsers (Chromium, Firefox and Opera) have only 475-542 mW average power consumption. Moreover, Firefox for Android puts its major threads onto big cores: 84% of its Gecko rendering engine thread and 60% of its Compositor thread on big cores.

Based on these findings, we make the following guidelines for reducing the energy cost in web page loading: 1) the web resource processing should adapt to download speed to become energy efficient; 2) the content painting should adapt to visible changes on the screen to save energy; and 3) the OS scheduler should be aware of QoS of threads to make energy-friendly scheduling decisions. Next we show how we develop techniques for energy-efficient web page loading according to these design guidelines.

4. ENERGY-EFFICIENT PAGE LOADING

In this section, we describe the three techniques aiming to better balance the energy cost and performance in mobile web page loading. We carefully design the techniques to control their impact on the page loading time.

4.1 Network-Aware Web Resource Processing

The choice of how often to conduct web resource processing influences performance and energy efficiency. Frequent resource processing may come with energy inefficiency, since it can yield marginal progress while consuming a significant amount of energy. As shown in 3.1, the Renderer process may conduct the web resource processing over each small amount of data (e.g., less than 3 KB in 2 Mbps download speed). This entails the need to perform batch processing of web resources to become more cost-effective. However, this may delay the processing of web resources and increase the page loading time. For example, an HTML resource typically contains links to other web resources such as images and JavaScript libraries. While the Renderer process delays the processing of the HTML resource, the Browser process cannot proceed to download the embedded resources from the network. So it calls for a good strategy to coalesce resources for energy-efficient resource processing yet without increasing the page loading time.

To this end, we propose to design the web resource processing adaptive to the speed of downloading web resources, which we refer to as *Network-aware Resource Processing* (NRP). A basic underlying principle is that the faster downloading, the larger batch size for better energy-efficiency should be. That is, we have a smaller batch size for slower downloading so as not to introduce excessive delays in web resource processing. We enforce the Browser process to buffer the data received from the network and notify the Renderer process only if the buffered data amount is larger than a threshold. This way, we can reduce the times of web resource processing as well as the number of IPCs involved, and thus can reduce the energy cost.

To balance the trade-off between resource processing delay and energy efficiency, we design the buffer threshold `buf_threshold` and the total buffer size `buf_size` to adapt to the resource downloading speed `net_goodput` as follows.

$$\begin{aligned} \texttt{buf\_threshold} &= \alpha * \texttt{net\_goodput} \\ \texttt{buf\_size} &= \beta * \texttt{buf\_threshold}, \end{aligned}$$

where `net_goodput` is estimated using the exponentially weighted moving average (EWMA), a linear history-based estimator, with a smoothing factor of 0.3, as described in [25, 32]. The parameter α indicates the maximum latency that a piece of data can experience in the buffer and is set to 0.5 second, at which we find a balance between energy saving and resource processing delay. The parameter β controls the total buffer size. It is increased when the buffer is overused and decreased when underused. When the buffer is full, it is increased linearly and decreased when average buffer usage of the last 20 samples is less than 50%. We cap the maximum buffer size to 2 MB.

The additional parameter `buf_size` is designed to minimize the total memory usage. To process each web resource, it requires allocating a dedicated buffer of size `buf_size`. As there may be many web resources in loading a web page, the total memory usage may be high if the buffer size is large. With our approach, we design the `buf_size` to be small while big enough to allow the downloading to keep up with the processing of resources.

Ideally, NRP should adapt directly to the user-perceived content changes on the screen. That is, the buffering should be adaptive to how fast the web content is displayed on the screen. However, detecting the changes of the displayed content from the received data is difficult and imposes high overhead because of dynamic content like JavaScript. Therefore, we choose the network throughput (the rate of receiving data) as an indirect but low-overhead indicator of the changes of the displayed content. By controlling the buffer threshold according to the throughput, we balance the energy cost and the latency of the web-resource processing.

To further reduce the impact of NRP on page loading time, we do not apply the buffering to critical resources that may delay the downloading of other resources. Such critical resources include the top level HTML resources (called *main resources* in Chromium) of the main frame and the sub-frames or iframes of a web page. Those resources are processed immediately after they are received from the network. Furthermore, we do not buffer the last chunk of a web resource even if the chunk size is smaller than `buf_threshold`, to prevent a small last data chunk from staying in the resource buffer for a long time. This is achieved by detecting if the response completed or not, which is performed by a network stream parser in the browser. Therefore, it is applicable to dynamic content as well. If the response completed, we notify the Renderer process starts to process all the received data immediately instead of waiting for reaching `buf_threshold`.

4.2 Adaptive Content Painting

Content painting comes with a trade-off between user experience and energy overhead. Although a high frame rate can provide very smooth user experience, it can incur a high overhead to GPU and CPU to render web pages and, when combined with high resolution, consume a lot of energy [33]. As shown in Section 3.2, each individual content paint may introduce a different degree of change on the screen. As the majority of content paints generate a zero or very small visible screen change, multiple content paints can be aggregated together to save energy without compromising user experience. Motivated by this, we design the content painting to be adaptive to the visible changes on the screen, which we refer to as *Adaptive Content Painting* (ACP), as follows.

We introduce a new parameter, called `paint_rate`, that limits the rate of content painting and dynamically adapts to the content changing speed so that we can aggregate content painting to save energy while preserving user experience. For a given value of `paint_rate`, ACP enforces that the actual content painting rate will never exceed the value of `paint_rate` (the actual rate can be lower than `paint_rate`). The `paint_rate` parameter ranges from a minimum painting rate to a maximum one. The initial value of `paint_rate` is set to the minimum value. The `paint_rate` parameter increases when the content changes fast and vice versa. After the Compositor thread paints the changed content to the screen, if the web page continues changing and needs to update the screen to reflect the change, we increase the parameter linearly by one to the maximum value. On the other hand, we decrease the parameter to the minimum value when the content of web page display stops changing after the Compositor delivers all the changes to the screen. In Section 5 we describe how we detect the speed of the content changes in Chromium.

We cap the maximum painting rate to 10 frames/s to balance the energy cost and the smoothness of content display. We argue that 10 frames/s should be smooth enough because 1) during page loading time, the web content is only partially displayed and thus the impact on user experience is small (especially on smartphones with small screens); and 2) the rate of content changes on the screen is often low as shown in Section 3.2. Furthermore, existing research [39] shows that a delay of up to 100 ms is typically not perceivable. Therefore, we believe that 10 frames/s (i.e., 100 ms

per frame) provides a good trade-off between user experience and energy saving. We set the minimum painting rate to 2 frames/s to save more energy when the content changes slowly.

Ideally, ACP technique should quantify the visual changes between painted frames to adapt with the degree of changes of content. However, doing so requires comparing consecutive frames pixel by pixel, which imposes heavy computation and thus a high energy cost. Therefore, we use a light-weight approach of linearly increasing the `paint_rate` parameter, without any extra computation cost.

Furthermore, the content painting should be aware of user interactions. A high painting rate should be used when the user touches the screen, to ensure smooth user experience. Therefore, in our implementation, we also detect user inputs and stop rate throttling when the user touches the screen.

The two techniques of NRP and ACP are not completely independent. NRP batches the processing of resources and thus also slows down the content painting. Similar to NRP, ACP also reduces the number of IPCs between the Browser process and a Renderer process to save energy.

4.3 Application-Assisted Scheduling

To better leverage big.LITTLE architecture to save energy, we propose to leverage internal knowledge of browsers for energy-efficient scheduling. Instead of letting the OS task scheduler make all the scheduling decisions by passively observing the load of threads, we allow browsers to decide whether a thread should run on a big or little core. Browsers know much more information about their threads than the OS scheduler, e.g., what type of tasks the threads do, how important the threads are, how long finishing time a thread can tolerate, the semantics of the threads, and the relationship among them. Therefore, browsers may make better decisions on assigning threads to big or little cores. We call this kind of technique *Application-Assisted Scheduling* (AAS).

We design the AAS technique adaptive to Quality of Service (QoS), as QoS is important to user experience. The QoS is estimated by the frame rate of the browser. The QoS requirement is dynamically adjusted to the changing value of `paint_rate`. If ACP is not working, then the requirement is fixed as the maximum limit on painting rate (i.e., 10 frames/s). AAS first allocates a set of threads related to the QoS on little cores and monitors the current frame rate. When the current frame rate is lagging behind the QoS requirement, AAS considers it the violation of QoS. For example, if the value of `paint_rate` is 5, the QoS is defined as painting each frame within 200 ms. If the browser fails to paint each frame within 200 ms, AAS decides that the QoS is violated and thus migrates the threads related on big cores. When the current frame rate becomes exceeding the QoS requirement in a stable manner, AAS brings those threads back to little cores. AAS makes such a decision when it finds the cumulative gap between the current frame rate and the QoS requirement over a window is no less than a threshold. Thus, while the QoS is being satisfied, its corresponding threads keep remaining on the little cores to save energy without compromising user experience.

To better utilize little cores to save more energy, we design AAS to move a thread from a little core to a big core in a conservative way of using a time window of three seconds, and move a thread from a big core back to a little core using a smaller time window of one second.

ACP adjusts `paint_rate` limit while AAS only monitors the actual frame rate (frame painting time) and schedules threads to satisfy the limit. Since the `paint_rate` sets only the maximum painting rate rather than a fixed painting rate, there is no strict circular dependency between the AAS technique and the frame rate.

Thread migration between little cores and big cores in AAS is generic to all applications. The AAS technique dynamically changes the affinity of the QoS-related threads, and thus does not require hard assignments of specific threads to cores. However, which threads should be managed by AAS and how to define the QoS are application specific. In Section 5 we show to which threads of Chromium we apply the AAS technique.

5. IMPLEMENTATION

We have implemented the three techniques on Android by modifying the source code of Chromium version 38, the latest stable version of Chromium when we conducted the work of this paper. We use the Content module (ContentShell.apk build target) that has the core code upon which Android Google Chrome browser is built [5]. Compared to Content Shell, Chrome has additional features such as auto fill, translation, and account setting synchronization.

In total, we add about 1,200 lines of code into various modules of the Chromium web browser. No rooting or modification to websites and OS kernel are needed. As we focus on page load time, we enable the techniques only during loading a web page, we disable the techniques after the web page loading is finished.

Network-aware Resource Processing. We implemented adaptive resource buffering by modifying the *BufferedResourceHandler* class in the Resource Handler chain in the Browser process. We revise the resource handler so that it keeps buffering until the received data size reaches the buffer threshold or the web-resource fetching has completed. In addition, the revised handler decides the resource type embedded in the HTTP requests. For top-level HTML resources and nested frames, it does not perform resource buffering to avoid delaying the loading of the web resources embedded in the HTML pages.

The Buffered Resource Handler can be paused when the rendering engine has not finished processing the resource and there is no space left in the resource buffer. When it resumes, it continues receiving data.

Adaptive Content Painting. We control content painting rate by controlling the rate of VSync callbacks. To do it, we wrote a shim layer on the *WindowAndroid* class which represents an Android activity window and holds a VSync monitor. This shim layer intercepts actual VSync callback from the VSync monitor and delays them by multiple of VSync periods before sending to the Compositor thread.

The content changing speed is determined by looking at the VSync request from the Compositor thread. If the rendering engine needs to update the layer tree and thus the Compositor thread still needs a VSync to perform a paint after all the changes have been painted, it indicates that the content have been changed since the last time. On the other hand, when the Compositor thread stops needing VSyncs to perform another paint, this indicates that the content does not change further. At this point, we stops requesting VSync from Android and decrease the frame rate limit to the minimum value.

Throttling the content painting rate may cause apparent lags when the user touches the screen. To solve this problem, we detect the user inputs to enable and disable content painting rate dynamically. We do this by instrumenting the *ContentViewCore* class which represents the view of a web page. Thereby, user will not feel any animation lagging when he interacts with the screen during page loading.

Application-Assisted Scheduling. We implement AAS by extending current Chromium threading management module to support changing thread CPU affinity dynamically and systematically.

On Android OS, thread CPU affinity is altered by system call *sched_setaffinity*, which does not require root privilege. Since default bionic C library level 19 on Android 4.4 does not provide native *sched_setaffinity* API, we directly invokes its assembly language interface by using *syscall* function together with its system call number and a CPU core list. We monitor the frame rate by using the layer tree debugging feature in Chromium which provides frame rate counter and paint time.

According to the analysis of the web browser architecture in Section 2, AAS schedules the Compositor Raster Worker and Async Transfer Thread. These threads affect the rate of frames painted to the screen by the graphic pipeline so directly relate to the user experience. They also have high CPU workload so scheduling them to little core will have high reduction to energy consumption of the browser.

6. EVALUATION

In this section, we evaluate our implementation by answering the following questions: 1) how much total energy we can save; 2) how much energy is saved by each of the three techniques; and 3) how much the techniques affect page load time and user experience.

6.1 Experiment Setup

Devices. We conduct experiments on two variants of the Samsung Galaxy S5 smartphones: S5-E and S5-S. S5-E is the SM-G900H model that uses Samsung Exynos 5422 SoC with 4 Cortex-A15 big cores and 4 Cortex-A7 little cores and Linux kernel 3.10. S5-S is the SM-G900K model that uses a quad-core Qualcomm Snapdragon 801 SoC and Linux kernel 3.4. Both smartphones run Android Kitkat 4.4.2 and Chromium version 38. We use a Monsoon Power Monitor tool [29] to measure the total system energy consumption of the smartphones.

To minimize measurement noise, we remove all unnecessary applications and background services, disable irrelevant hardware components such as camera, GPS and other sensors, and turn off all wireless network interfaces except WiFi. The screen brightness is set to the lowest level.

Testbed and network conditions. For repeatable experiments and to avoid the noises of unstable cellular network and unpredictable response of web servers, we build a testbed to load websites from a local emulated server. The Linux server records and replays responses of websites using the *Web Page Replay* [13] tool. The smartphones load the web pages from the server through a local WiFi network. The server emulates a 3G cellular network using the *dummynet* network emulator [7]. Based on previous measurement studies [40, 26, 1, 27] on 3G cellular networks, we use the following parameters to emulate typical cellular network conditions: 2 Mbps downlink bandwidth, 1 Mbps uplink bandwidth, and a 120 ms round-trip time (RTT).

Data set. We evaluate our techniques using the top 100 websites in the U.S. according to Alexa [2] in May 2014. The 100 websites include diverse websites including sophisticated ones with many images and simple ones with mostly textual contents. Among them, 88 websites have a mobile-friendly design. By default, we use the homepage of the websites in our experiments. For the websites requiring a user login, we record and replay the response of the website after logging in, or choose a representative public web page from the website. For the websites with a so trivial homepage (such as search engine *ask.com*), we choose a non-trivial, representative, second-level web page.

Automation tool. We develop a tool to automate the experiments and data collection. The tool has two modules, one running on a smartphone and the other running on a Windows PC that runs the Monsoon Power Monitor software to collect energy consumption data. On the smartphone, the tool is a module on the content shell layer (e.g., ShellManager class) of Chromium. It takes a list of URLs and controls Chromium to visit the URLs one by one. Before visiting each URL, it clears the cache of Chromium. During loading a web page, it collects the data about the page loading such as the page load time. After the experiments are finished, it transfers all the collected data to the PC. On the PC, the tool controls the Monsoon software to record the energy consumption data. The tool also precisely synchronizes the time of the smartphone and the PC using a synthetic falling edge on the power trace. Therefore, we can align the power trace data collected on the PC to the page loading data on the smartphone.

Page load time. Page Load Time (PLT) measures the time since a web page starts loading, i.e., the Navigation Start event, until the page finishes loading, i.e., the Load Event End event [24, 23, 22]. We revise Chromium to record the timestamps of the Navigation Start event and Load Event End event, and thus our automation tool can calculate the PLT of each single web page. We measure cold page loading time with the browser cache and DNS cache cleared.

Unless otherwise stated, we repeat each experiment for at least 5 times. Measurement outliers are detected and removed by applying the modified Z-score [28] on page load time values of each configuration combination. We add more experiments if the number of remaining samples after removing outliers is too small. Each point on the cumulative distribution function (CDF) graphs on this section presents the average for each website. For estimating variation, we calculated 95% confidence intervals for each average but do not present here because most of the confidence intervals are less than 5% of the average values.

6.2 Energy Saving

Total energy saving. We first evaluate how much total energy saving can be achieved by all the three techniques together. Figure 5 shows the result. On S5-E, the average total energy saving in loading the 100 websites is 24.4%, ranging from 0.02% to 66.5%. 62 websites have an energy saving of 15%-45%, and 7 websites have an energy saving of more than 50%. These results demonstrate that our techniques are able to significantly reduce the energy cost of web page loading. We should also be able to achieve this high energy saving on Chromium-based web browsers such as Google Chrome and Opera Mobile. The most energy saving (66.5%) website is *infusionsoft.com* that has a heavy slide show of high resolution images with a fading transition effect. The least energy saving websites are the ones with simple text content such as *usps.com*.

The S5-S smartphone has a smaller energy saving than the S5-E smartphone. On S5-S, the average energy saving is 11.7%, ranging from 0.21% to 57.1%. 66 websites have an energy saving of 5%-30% and only two websites have an energy saving more than 50%. The reason is that S5-S does not support big.LITTLE and thus cannot benefit from application-assisted scheduling which is the most energy saving technique as we will show later in this section.

The above energy savings are measured in terms of the total system energy consumption, *including the energy cost of the screen*, because the web browser does not load web pages with screen off. Therefore, in order to evaluate the impact of power of the screen, we use a separate experiment. We run the experiment twice with the screen turned on and off. The screen power is then measured as the system power difference between the two different types of experiments. The two smartphones use the same type of screen and the screen power is 186 mW at the lowest brightness level. Exclud-

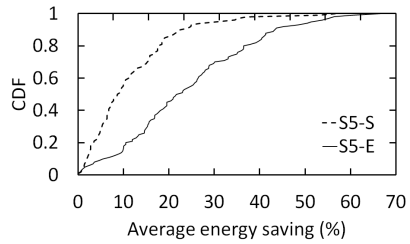


Figure 5: Total energy saving

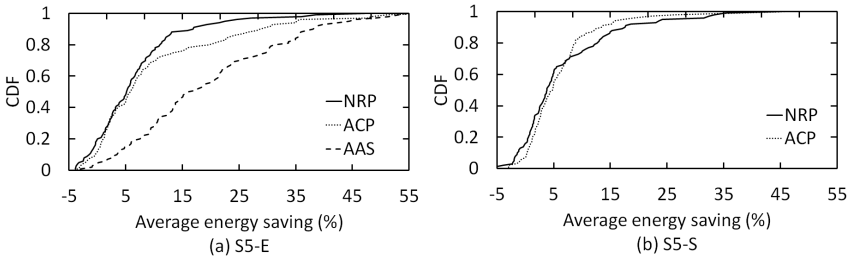


Figure 6: Energy saving of individual techniques

ing the energy cost of screen, the average energy saving increases from 24.4% to 26.6% on S5-E and from 11.7% to 13.0% on S5-S.

Energy saving of individual techniques. We next evaluate to what extent each technique can save energy consumption. Figure 6 shows the results when only one of the three techniques is enabled. NRP, ACP, and AAS stand for energy-aware resource buffering, aggregated content painting, and application-assisted scheduling, respectively.

On S5-E, the most energy saving technique is AAS, reducing the average energy consumption by 19.5%. The other two techniques are also useful, with an average energy saving of 10.2% for NRP and 6.8% for ACP, respectively. Only NRP and ACP can be used on S5-S and have average energy savings of 6.8% and 6.3%, respectively. If we exclude the energy cost of screen, on S5-E, the average energy savings for NRP, ACP and AAS are 11.1%, 7.4% and 21.1%, respectively; and on S5-S, the average energy savings of NRP and ACP are 7.6% and 6.9%, respectively. These results demonstrate that all the three techniques are effective in reducing the energy consumption of web page loading.

It is worth to note that, for some websites, enabling only one of the techniques may slightly increase the system energy cost. For example, on S5-E, if only the AAS technique is enabled, four websites have increased energy costs, ranging from 0.28% to 3.0%. The maximum energy cost increases on S5-E and S5-S are 3.8% and 5.6%, respectively. However, with all the three techniques (two on S5-S), it never increased the energy consumption for all the 100 websites.

6.3 Page Load Time and User Experience

Next we evaluate how much PLT is increased by the combination of the three techniques and by each individual technique, and study the impact of the techniques on user perceived experience using Speed Index [10] and a user study.

Total page load time increase. Our techniques impose minimal extra PLT. Figure 7 shows the results on the two smartphones (all the three techniques are enabled on S5-E and only two techniques are enabled on S5-S). On S5-E, the average PLT is decreased by 0.38% (or decrease 29 ms in terms of the absolute PLT), ranging from -8.11% to 6.38%. In fact, 55 websites have a decreased PLT and 93 websites have PLT increases less than 3%. In terms of the absolute PLT, 94 websites have PLT increases less than 0.2 seconds. On S5-S, the average PLT increase is almost zero, 0.01%, ranging from -4.1% to 4.5%. 51 websites have a decreased PLT, and 94 websites have a PLT increase less than 3%. In terms of the absolute PLT, the average PLT is increased by 6.7 ms, and 93 websites have a PLT increase less than 0.2 seconds. We argue that it is worth to trade this small PLT increase for the significant energy saving. In particular, we are indeed able to decrease the PLT of about half of the 100 websites.

Page load time increase of individual techniques. We also measure the PLT increase caused by each individual technique. Figure 8 shows the results. As one may expect, each technique alone also introduces minimal extra PLT. On S5-E, AAS introduces the highest average PLT increase but the increase is only 0.23% and it even decreases 22.8 ms in terms of absolute PLT. The PLT increase for NRP and ACP are 0.04% (-2.6 ms) and -0.45% (-49.9 ms), respectively. On S5-S, the PLT increase is 0.76% (56.2 ms) for NRP and -0.25% (-24.5 ms) for ACP, respectively. Compared to the other two techniques, ACP has the smallest impact on PLT as it decreases PLT on both phones.

Configuration	Average PLT increase		Average energy saving (%)	
	Relative (%)	Absolute (ms)	With screen	No screen
All	-0.38	-29.0	24.4	26.6
NRP	0.04	-2.6	10.2	11.1
ACP	-0.45	-49.9	6.8	7.4
AAS	0.23	-22.8	19.5	21.1

Table 2: Energy saving and PLT increase on S5-E

Configuration	Average PLT increase		Average energy saving (%)	
	Relative (%)	Absolute (ms)	With screen	No screen
All	0.01	6.7	11.7	13.0
NRP	0.76	56.2	6.8	7.6
ACP	-0.25	-24.5	6.3	6.9

Table 3: Energy saving and PLT increase on S5-S

Table 2 and Table 3 summarize the average energy savings and PLT increase on S5-E and S5-S, respectively. It is worth noting that the three techniques are not completely independent, and thus the total energy saving is not equal to the sum of the energy savings of individual techniques.

Impact on visually complete progress. In order to evaluate the impact of the techniques on the visual loading progress of web pages, we measure the Speed Index [10] in addition to the page load time. The metric measures how quickly the page content is visually populated, and is calculated as in the following formula:

$$SpeedIndex = \int_0^{end} (1 - \frac{VC}{100}),$$

where end is the end time in milliseconds and VC is the percentage of visually complete. Lower Speed Index is better because the user will see the web page content painted earlier and thus faster visual loading progress.

We use Telemetry tool to get visual progress from paint events exposed by Blink through the DevTools timeline. Compared to getting visual progress from video capture, this method deals with pages that update and change better [10, 12]. Moreover, it worked more reliably on the websites in our dataset specially ones with animations.

Our techniques increase the metric by 1.8% on average, so they delay the page loading progress only slightly. Figure 9 shows the increase of Speed Index of the All configuration compared to the

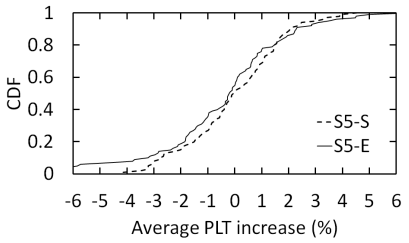


Figure 7: Total PLT increase

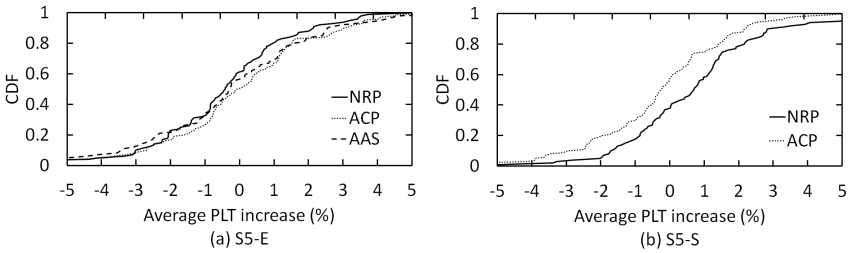


Figure 8: PLT increase of individual techniques

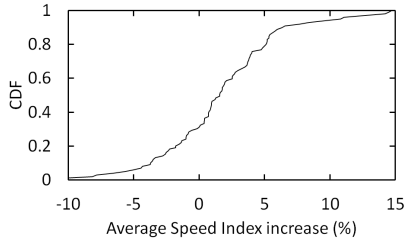


Figure 9: Impact of All configuration on Speed Index metric

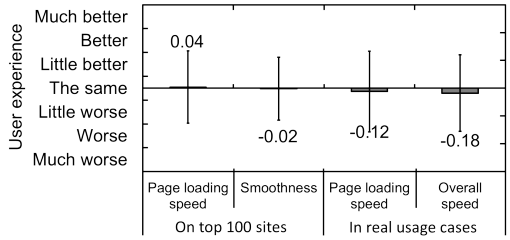


Figure 10: User perceived experience on our revised Chromium compared to the default Chromium

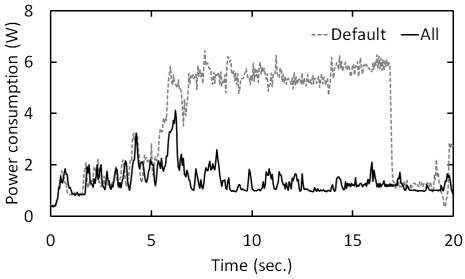


Figure 11: Power consumption in loading infusionsoft.com with All configuration

Default. We report the results for 99 websites, except *conduit.com* which contains a continuously rotating image and for which the Telemetry tool reported an abnormal decrease of 42% by our techniques.

User perceived experience To further evaluate the impact of the increased PLT on user perceived experience, we conduct a user study to collect real user feedback on our techniques on real-world environments. Therefore, we recruited 18 undergraduate students in our university, 11 males and 7 females and performed two tests using the S5-E smartphone on a real-world WiFi network. In the first test, we use the default Chromium and our revised Chromium to load 10 web pages randomly selected from the top 100 websites. Each website is loaded using either the default Chromium or our revised Chromium browser. We show the web page loading to the users but do not let them know which Chromium version is used. User can ask to repeat loading any web page as many times as he/she wants. We then ask them whether our revised Chromium is better or worse than the default Chromium in terms of page loading speed and smoothness, using a seven-level scale (from 3 to -3 for much better, better, little better, the same, little worse, worse, and much worse, respectively). In the second test, we let the users do web browsing in their favorite way, using each version of the browser for 5 minutes, respectively. The users also do not know which Chromium version being used. This test includes not only page loading but also page reading and users interactions (e.g., scrolling). Then we ask them to compare the two browsers again using the same seven-level scale, in terms of page loading speed and the overall speed.

Figure 10 shows the results. For the first test, 4 of 18 (22%) users told that it is difficult to tell which browser is faster. 10 (55%) users even gave a higher score on our revised browser. For smoothness, most of the users could not tell the difference. The average scores of page loading speed and smoothness are almost zero: 0.04 (standard deviation(stdev) 1.30) and -0.02 (stdev 1.14), respectively. In the second test of real usage, we get similar results: 7 (39%) users told that our revised browser is faster in page loading, and 5 (28%) users told that our revised browser is faster in overall speed. The average scores of page loading speed and the overall speed are also near zero: -0.12 (stdev 1.45) and -0.18 (stdev 1.38), respectively. These results demonstrate that our techniques impose a minimal impact on user experience and such a small impact is hardly perceivable by the users. After the tests, we asked the users how they would like to use our techniques. 13 (72%) of them would always use our techniques and 5 (28%) of them would use when the battery level is low.

6.4 Energy Saving Analysis: Case Study

To investigate further how our proposed techniques reduce energy consumption, we present a detailed analysis for *infusionsoft.com* as a case study since it is the most energy saving website for each technique. In essence, our techniques decrease the total energy consumption by reducing system power and CPU utilization. We examine the results obtained on the S5-E smartphone.

System power reduction. As our techniques introduce a minimal increase of PLT while reducing total consumed energy, what we actually reduce is system power consumption. Figure 11 shows the power traces in loading *infusionsoft.com*, the website that has the highest energy saving (66.5%). Our techniques are able to significantly reduce the system power. For the time period of 5.0 sec - 17.0 sec in the figure, the average system power in using the default Chromium is 5.28 W; with our techniques enabled, the aver-

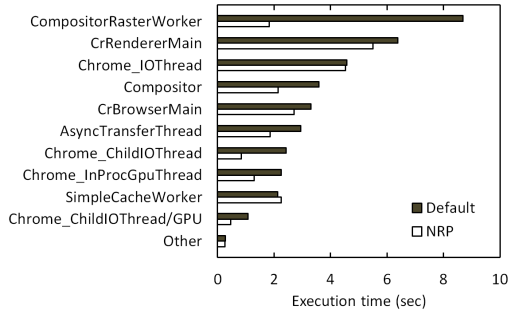


Figure 12: Thread CPU time in loading infusionsoft.com with NRP configuration

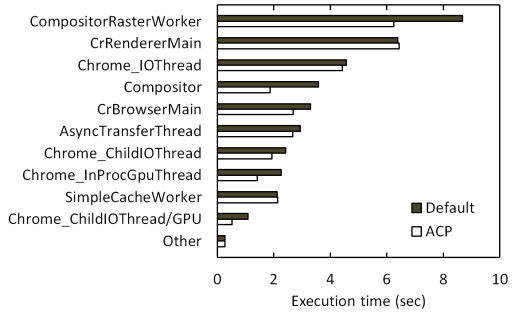


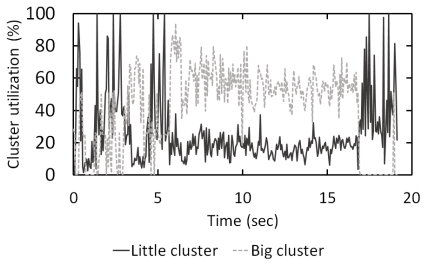
Figure 13: Thread CPU time in loading infusionsoft.com with ACP configuration

age system power is reduced to 1.44 W. This significant reduction of system power comes from the reduced CPU utilization, as shown below.

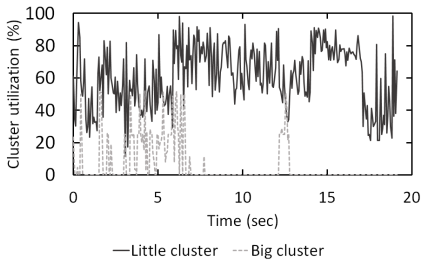
CPU utilization reduction of NRP. Figure 12 shows the CPU time of the main threads in loading *infusionsoft.com*, using the default Chromium and using our revised Chromium with only the NRP technique enabled. When NRP is enabled, the CPU time of the *Chrome_ChildIOThread* thread is significantly reduced, from 2.42 seconds to 0.84 seconds, or a reduction of 65.3%. This is because NRP is designed to the reduce IPCs between the Browser process and the Renderer process. NRP also reduces the CPU time of the *CompositorRasterWorker* thread (from 8.68 seconds 1.83 to seconds, or 78.9%), *Compositor* thread (from 3.58 seconds to 2.14 seconds, or 40.2%), *Chrome_InProcGpuThread* thread (from 2.26 seconds to 1.29 seconds, or 42.9%), and *AsyncTransferThread* thread (from 2.94 seconds to 1.86 seconds, or 36.7%). This is because increasing data size for each processing reduces the number of changes to layers of a web page and thereby the number of content rendering. This way, it significantly reduces the CPU processing time of the rendering engine and graphic processing.

CPU utilization reduction of ACP. Figure 13 shows the results with only the ACP technique enabled. When ACP is enabled, the CPU time of the following four threads are significantly reduced: for *Compositor* thread, from 3.58 seconds to 1.87 seconds (47.8%), for *Chrome_InProcGpuThread* thread, from 2.26 seconds to 1.41 seconds (37.5%), and for *CompositorRasterWorker* thread, from 8.68 seconds to 6.25 seconds (28.0%). These results demonstrate the effectiveness of ACP in reducing the processing cost of the graphic processing pipeline.

CPU utilization reduction of AAS. AAS reduces big cores workload significantly while increasing utilization of little cores for energy efficient processing. Figure 14 shows the utilization of little and big clusters when loading *infusionsoft.com*. In default



(a) Default configuration



(b) AAS configuration

Figure 14: Utilization on clusters of application-assisted scheduling

Configuration	Average PLT increase		Average energy saving (%)	
	Relative (%)	Absolute (ms)	With screen	No screen
All	1.69	180	10.5	11.9
ACP	-0.46	-36	9.5	10.5
AAS	2.00	195	3.3	3.9

Table 4: Energy saving and PLT increase on Firefox

Chromium, the average utilization of little and big cores are 25.2% and 40.9%, respectively. Using AAS, the average utilization of little cores increases to 60.1% while the average utilization of big cores decreases to only 6.1%.

6.5 Energy Saving on Other Web Browser

In order to demonstrate applicability of techniques on other web browsers, we implement the ACP and AAS techniques on Firefox for Android open source web browser. Among the top 100 websites, our experiments were performed on 72 websites that use HTTP protocol. This is because it is problematic for Firefox to establish HTTPS connections to the Web Page Replay server which uses a self-signed encryption certificate. We use Firefox version 38, the latest version at the time of writing this paper. For ACP, the implementation is only 50 lines of code in the CompositorParent class which is responsible for graphical composition in Firefox. The implementation of AAS involves changes to thread management module. Because the NRP technique requires considerable implementation of a whole new buffer layer in the network stack of Firefox, we leave it as future work.

With the ACP and AAS techniques, the revised Firefox web browser saves a non-trivial amount of energy while affect page load time minimally. On S5-E, both techniques save 10.5% average system energy saving while increasing 1.69% (180 ms) PLT. For each technique, average system energy saving is 9.5% for ACP and 3.3% for AAS. The PLT increase is -0.46% for ACP and 0.2% for AAS. The results are shown in Table 4. The reason the AAS technique does not save much energy on Firefox is that the AAS technique attempts to put Compositor thread to little cores. However, although the thread has higher CPU utilization than other threads, it still has light CPU load. Therefore, putting Compositor thread onto a lit-

Device	Average PLT increase		Average energy saving (%)	
	Relative (%)	Absolute (ms)	With screen	No screen
S5-E	-0.06	-3.5	21.8	23.4
S5-S	-0.92	-47.3	6.0	6.5

Table 5: Energy saving and PLT increase on a fast network

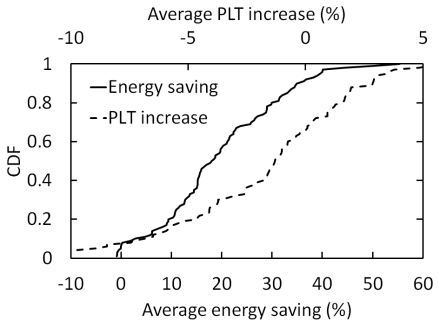


Figure 15: Energy saving and PLT increase in hot loading

tle core does not save much energy compared with the case where the Compositor thread is running on a big core at the least possible frequency.

6.6 Energy Saving on Fast Networks

We also evaluate how much energy our techniques can save on fast networks such 4G cellular and WiFi. We emulated a fast network with 20 Mbps downlink bandwidth, 10 Mbps uplink bandwidth, and 50 ms RTT. Table 5 shows that our techniques still save 21.8% and 6.0% of system energy on S5-E and S5-S, respectively. Excluding screen energy, our techniques save 23.4% and 6.5% on S5-E and S5-S, respectively. The PLT is even decreased by 0.06% (S5-E) and 0.92% (S5-S). This decrease of PLT may come from two reasons: 1) when the network is faster, the extra latency introduced by NRP becomes smaller; 2) as AAS makes better use of little cores, it leaves more capacity of big cores to the rendering engine and thus makes content rendering faster.

6.7 Energy Saving in Hot Loading

We evaluate the performance of our techniques in case of web page loading with cached content. We test on the hot web page loading in which the web browser loads the web pages again immediately after a cold load. Thereby, many resources are still in the web browser cache and DNS lookup results are also still in the local DNS cache of the OS. We use the emulated WiFi network with 2 Mbps downlink, 1 Mbps uplink bandwidth and a 120 ms RTT.

Our techniques are still effective in reducing the system energy consumption in hot loading. The results of the All configuration on S5-E are shown in Figure 15. The average system energy is reduced by 19.6% and average page load time is decreased by 1.73% (or 67 ms in terms of absolute page load time). The decrease of page load time can be explained similarly as on fast networks.

6.8 Energy Saving on 3G Network

We evaluate the energy saving of our techniques when running on a 3G network, given that the power consumption profile of a cellular network interface is different from WiFi [36]. For repeatable experiments, web pages are loaded from our Web Page Replay server. The server throttles bandwidth by 2 Mbps download and 1 Mbps upload while not adding any delay. Given the 3G (HSPA+) network is fast enough, the bandwidth is limited at the server side. The delay is not controlled and determined by the 3G network. We

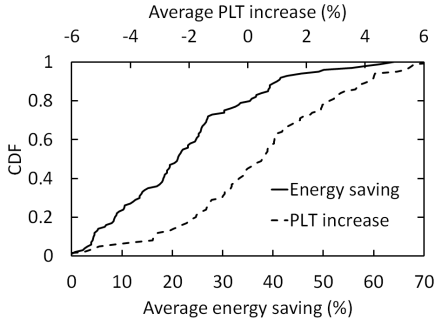


Figure 16: Effectiveness of All configuration on a 3G network

use the 3G network provided by KT Corporation, and the phone has only its 3G interface turned on (its WiFi interface is off).

Figure 16 shows the results of All configuration on S5-E. Our techniques have average system energy saving of 22.5% and average PLT increase of 0.41% (27 ms in absolute page load time). Average system energy saving when using a cellular network interface is still high although it is about 2% lower than when using a WiFi network interface (24.4%). Given the amount of downloaded data for each web page is not large (e.g., 1.1 MB on average in our dataset), the processing on the CPU generally dominates the power consumption of the whole system. The measured energy is the total system energy which includes the 3G tail energy during web page load time but does not include the tail energy after the page loading finishes.

7. DISCUSSION AND FUTURE WORK

Apply the proposed techniques to other applications. Other browsers and applications may use the three proposed techniques to improve their energy efficiency. Two techniques, NRP and ACP, are about general processing of web contents and thus may benefit other browsers. In particular, the Chromium-based web browsers such as Opera Mobile and WebKit-based web browser such as Safari may directly benefit from the techniques.

Furthermore, many mobile apps are designed to access web contents (e.g., a native app of a news website) and some of them are even directly built on top a browser control (e.g., the Chromium-based WebView class on Android). They are essentially customized browsers and thus may also benefit from our techniques. Furthermore, AAS is not specifically designed for browsers. The idea of leveraging application internals to make better scheduling decisions is general enough to be used by more applications. In particular, on big.LITTLE platforms, we see big potential of this technique and plan to investigate more on how to apply this technique to more applications. Furthermore, AAS technique is applicable to non-rooted Android since setting thread cpu affinity does not require the root priviledge. Also, current Android 5.0 also uses the same kernel version 3.10 on big.LITTLE architecture phones as on Android 4.4 so this technique will be also effective.

Reduce energy consumption after page loading. Although we focus on the energy consumption of page loading in this paper, it is possible to save more energy after page loading, i.e., when a user reads and interacts with a web page. In fact, the AAS technique may be applied to user reading time as well. We have measured the energy consumption of the top 100 websites with 10 seconds of non-interactive user reading time, and found that AAS is able to reduce the system energy consumption in the user reading time by 4.6% (5.6% excluding screen energy) with minimal user experience degradation. Furthermore, after page loading, the web con-

tents have been processed and thus we may apply more aggressive CPU power saving policies to save more energy. We plan to investigate more energy saving techniques for the time period after page loading.

Offload more graphic processing to GPU. As we have shown, the graphic-processing pipeline has heavy workload and is energy consuming, especially the rasterization. As GPU is more efficient in doing graphic tasks, it is possible to move the graphic processing currently done on CPU (such as Raster Worker) to GPU to save energy. In fact, recent experimental features of Chrome have demonstrated big improvement in reducing frame generation time [38], and thus has potential to reduce energy consumption. However, this technique is orthogonal to our techniques. We focus on optimizing IPCs overhead, granularity of content processing and painting, and task scheduling on big.LITTLE architecture.

Real deployment and field study. Although our in-lab experiments and user study have demonstrated the effectiveness of our techniques, they show few insights on how our techniques could perform in real usage. It is desirable to deploy our revised Chromium browser to end users and further evaluate its performance in more realistic and diverse network conditions and settings. We plan to conduct a long-term field study to collect real user feedback from daily use.

Re-design mobile browser architecture. Ideally, the architecture of mobile web browsers should be re-designed to have energy consumption as the first-class consideration at the very beginning. The web browser should be aware of the energy cost of each stage of the content fetching and processing pipeline, adapt to the user-perceived experience, and reduce the interaction overhead of different components. It should also better utilize new hardware capabilities such as big.LITTLE cores to save energy. We plan to investigate more on how systematic architecture re-design may help us further balance the energy consumption and user experience of mobile web browsing.

8. RELATED WORK

Energy-efficient mobile web browsing has been explored through various approaches. Thiagarajan *et al.* [41] examined the energy consumed to render individual web page primitives, such as HTML, images, JavaScript, and Cascade Style Sheets (CSS), and suggested guidelines for designing energy-efficient web pages, such as diminishing complex JavaScript and CSS and using JPEG images. Zhu *et al.* [46] introduced energy-efficient, latency-sensitive scheduling of web pages for mobile browsing over heterogeneous cores, according to statistical inference models that estimate page load time and energy consumption through the characteristics of web page primitives, HTML and CSS. Butkiewicz *et al.* [15] presented a linear regression model that predicts page load times according to the characteristics of web pages (e.g., number and size of images) and web servers (e.g., number of servers/origins). Chameleon [20, 21] renders web pages with energy-efficient color schemes on OLED mobile systems under user-supplied constraints. Qian *et al.* [35] provided a deep analysis of how mobile web browsing uses wireless network resources and offered guidelines for energy-efficient browsing, such as decomposing a tall web page into several smaller subpages and reducing JavaScript-triggered delayed or periodic data transfers in cellular networks. Zhao et al. [45] reorganized computation stages in web browsers and predicted user reading time in order to quickly put the radio interface into the power-saving IDLE state on 3G based phones. Our work can be differentiated from the above approaches in that we leverage browser internals (e.g., process/thread structure, resource fetching/processing pipelines) for energy-efficient browsing, while the others focus on the characteristics of web pages (e.g., primitives, colors, network accesses). Thereby, we believe our work can be integrated with others to improve energy efficiency further.

Mobile browsing performance optimization. Many techniques have been developed to reduce page load time without taking energy concerns into account. New web protocols such as SPDY [9] and HTTP 2.0 [8] are now under development to improve the performance of current HTTP protocol. Their key features include (i) multiplexing HTTP transactions into a single TCP connection and (ii) prioritizing some object loads over others (e.g., JavaScript over images). A recent study [42] found that SPDY can improve page load time significantly over HTTP 1.1 by largely benefiting from the use of a single TCP connection. However, they also showed that such a benefit can be overwhelmed by dependencies in web pages and browser computation, and suggested to restructure the page load process to reduce page load time. In addition to such infrastructure-related approaches, client-only approaches were developed with the benefit of easy deployment without infrastructure support. Wang *et al.* [43] showed that two popular client-only schemes, caching and prefetching, can be ineffective for mobile browsing, while speculative loading can be helpful to overcome their limitation. Ma *et al.* [30] first proposed and used a proactive approach for a comprehensive study on mobile web cache performance to identify the problems of unsatisfactory cache performance and revealed the root causes. Meyerovich *et al.* [31] introduced algorithms to parallelize web page layout and rendering engines to speedup browser computation.

Energy saving for mobile apps. Pathak *et al.* [34] examined the energy consumption of smartphone apps in a fine-grained manner, reporting some interesting findings such as third-party advertisement modules consume about two-thirds of energy spent in free apps. Xu *et al.* [44] studied energy saving on email client through various techniques, including reducing 3G tail time and decoupling data transmission from data processing. Some researchers focus on effectiveness of DVFS on mobile CPU. A recent study [17] considered energy saving on big.LITTLE architecture with a focus on integrating core offlining with frequency scaling, while paying no attention to determining on which type of cores (big or little) apps run for energy efficiency.

9. CONCLUSION

This paper presented three effective techniques to optimize the energy consumption of web page loading on smartphones. Two of the techniques, network-aware resource processing and adaptive content painting, are designed to address energy-inefficiency issues of the current mobile web browsers in its content processing and graphic processing pipelines. The third one, application-assisted scheduling, is designed to balance the trade-off between the energy saving and the QoS big.LITTLE platforms. We have implemented the proposed techniques on Chromium and Firefox, and conducted comprehensive evaluations using real-world websites and latest-generation smartphones. Experimental results and user study show that the techniques are able to significantly reduce the energy cost of web page loading and introduce hardly perceivable page load time increase.

ACKNOWLEDGEMENTS

This work was supported in part by ICT/SW Creative research program (2014-H0510-14-1008) of MSIP/NIPA, Korea, and Microsoft Research and by SW Computing R&D Program (2011-10041313) of MSIP/KEIT, Korea. We also thank our anonymous reviewers and shepherd for their insightful and constructive comments that helped us improve this paper.

10. REFERENCES

[1] 3G/4G wireless network latency: Comparing Verizon, AT&T, Sprint and T-Mobile in February 2014.

[2] Alexa, Top Sites in United States. http://www.alexa.com/topsites/countries/US.

[3] ARM big.LITTLE technology. http://www.thinkbiglittle.com.

[4] Blink. http://www.chromium.org/blink.

[5] Content module. http://www.chromium.org/developers/content-module.

[6] Differences between Google Chrome and Linux distro Chromium. http://code.google.com/p/chromium/wiki/ChromiumBrowserVsGoogleChrome.

[7] The dummynet project. http://info.iet.unipi.it/~luigi/dummynet.

[8] Hypertext transfer protocol version 2.0, draft-ietf-httpbis-http2-07. http://tools.ietf.org/html/draft-ietf-httpbis-http2-07.

[9] SPDY. http://www.chromium.org/spdy.

[10] Speed index. http://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[11] Telemetry. http://www.chromium.org/developers/telemetry.

[12] Visual progress - dev tools. http://www.webpagetest.org/forums/showthread.php?tid=12216.

[13] Web Page Replay. http://www.github.com/chromium/web-page-replay.

[14] WebKit. http://www.webkit.org.

[15] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proc. ACM IMC*, 2011.

[16] A. Carroll and G. Heiser. Mobile multicores: Use them or waste them. In *Proc. USENIX HotPower*, 2013.

[17] A. Carroll and G. Heiser. Unifying DVFS and offlining in mobile multicores. In *Proc. IEEE RTAS*, 2014.

[18] H. Chung, M. Kang, and H. D. Cho. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology, 2012.

[19] A. Cunningham. New Opera for Android looks like Opera, tastes like Chrome. http://arstechnica.com/information-technology/2013/05/new-opera-for-android-looks-like-opera-tastes-like-chrome.

[20] M. Dong and L. Zhong. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *Proc. ACM MobiSys*, 2011.

[21] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing (TMC)*, 2012.

[22] S. Dutton. Measuring Page Load Speed with Navigation Timing. http://www.html5rocks.com/en/tutorials/webperformance/basics, 2011.

[23] J. Glauner. Analyzing Website Performance at a Glance. http://www.stratigent.com/community/analytics-insights-blog/analyzing-website-performance-glance, 2013.

[24] U. Gundecha. *Selenium Testing Tools Cookbook*. Packt Publishing, 2012.

[25] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. In *Proc. ACM SIGCOMM*, 2005.

[26] J. Huang. *Performance and Power Characterization of Cellular Networks and Mobile Application Optimizations*. PhD thesis, The University of Michigan, 2013.

[27] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys*, 2010.

[28] B. Iglewicz and D. Hoaglin. *Volume 16: How to Detect and Handle Outliers*. 1993.

[29] Monsoon Solutions Inc. Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor.

[30] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie. Measurement and Analysis of Mobile Web Cache Performance. In *Proc. WWW*, 2015.

[31] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proc. WWW*, 2010.

[32] M. Mirza, J. Sommers, P. Barford, and Xiaojin Zhu. A machine learning approach to TCP throughput prediction. *Networking, IEEE/ACM Transactions on*, 18(4):1026–1039, 2010.

[33] 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 *HotPower*, 2014.

[34] 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 *Proc. ACM EuroSys*, 2012.

[35] F. Qian, S. Sen, and O. Spatscheck. Characterizing Resource Usage for Mobile Web Browsing. In *Proc. ACM MobiSys*, 2014.

[36] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. ACM IMC*, 2010.

[37] M. Rasmussen. sched: Task placement for heterogeneous MP systems. http://www.lwn.net/Articles/517250, 2012.

[38] R. Schoen. Wicked Fast (Performance investments). In *Chrome Dev Summit*, 2014.

[39] B. Shneiderman, C. Plaisant, M. Cohen, and S. Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Fifth edition, 2009.

[40] J. Sommers and P. Barford. Cell vs. wifi: On the performance of metro area mobile connections. In *Proc. ACM IMC*, 2012.

[41] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proc. WWW*, 2012.

[42] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proc. USENIX NSDI*, 2014.

[43] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proc. WWW*, 2012.

[44] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In *Proc. ACM MobiSys*, 2013.

[45] B. Zhao, Q. Zheng, G. Cao, and S. Addepalli. Energy-Aware Web Browsing in 3G Based Smartphones. In *Proc. IEEE ICDCS*, 2013.

[46] Y. Zhu and V. J. Reddi. High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems. In *Proc. IEEE HPCA*, 2013.