M.S. THESIS

A DVFS ENERGY-AWARE WEB BROWSER

BY

Richard G. Taylor

AUGUST 2013

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

A DVFS ENERGY-AWARE WEB BROWSER

BY

Richard G. Taylor

AUGUST 2013

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

A DVFS Energy-Aware Web Browser

동적 전압 주파수 스케일링 기반한 에너지 인지 웹 브라우저

지도교수 홍성수

이 논문을 공학석사 학위논문으로 제출함

2013 년 05 월

서울대학교 대학원

전기정보공학부

Richard Taylor

Richard Taylor 의 공학석사 학위논문을 인준함

2013 년 6 월

위 원 장 : 김태환

부위원장 : 홍성수

위 원 : 백윤흥

Abstract

Smartphone web browsing has surpassed that of desktop web browsing. Furthermore, users are placing higher demands on the functionality of their smartphone devices. As a result of these factors, web browser energy consumption has become a critical issue in keeping our smartphones awake. In this paper, we undertake improving the power consumption of a smartphone web browser through two innovative techniques. Firstly, we target the energy consumption of the network adapters by partitioning the workflow of the web browser into 2 phases; an IO bound loading and CPU bound rendering phase. This allows the device to switch the network adapter to a lower power idle state faster than that of the conventional web browser. Secondly, we focus on improving the energy consumption of the CPU by applying a DVFS algorithm to the device during a web page load. The DVFS algorithm exploits the partitioned architecture by running a low CPU frequency in the loading phase and an optimal frequency in the rendering phase. The optimal frequency is a CPU frequency that can perform the rendering of a web page in at least the same time as the default governor, but with lower energy consumption. To further unburden the resource constraints of the smartphone we use the processing power of the cloud to rapidly calculate the optimal frequency concurrently as the device is processing the loading phase. The

cloud uses a combination of web page characterization and regression models to accurately predict the optimal frequency for rendering the web page on the device. To demonstrate the effectiveness of our approach, we implement our proposal on an Android based Samsung Galaxy S2. Experimental results show that our approach can reduce the power consumption of a web page load by 12% in Wi-Fi and 14.5% in a UMTS 3G network environment, with no added latency to web page load times.

Keywords: Web Browsers, Smartphones, Android, Power Optimizations, DVFS

Student Number: 2011-24074

Contents

List of Figures

Chapter 1 – Introduction

Smartphones are getting faster all the time. Their batteries, however, aren't. As users are increasingly placing high demand on the functionality of their devices, poor battery life is driving down smartphone consumer ratings. Recent reports suggest that on a satisfaction scale from 1 to 5, the smartphone's battery life scored 2.99 points [6]; in comparison the phones' display quality scored 4.26 points. Even though battery life is among the top complaints of users, is it being overlooked by manufacturers?

In fact, at the present time, battery optimizations are among one of the hottest topics with manufacturers and carriers of smartphones. Complicating this fact though, is that smartphones keep getting faster; this year's top-of-the-line phones are likely to be twice as fast as last [20]. As a result of this, until new battery technologies become commercially available, energy aware applications and battery optimizations are indispensable to keeping smartphones powered in the near future.

The web browser is one of the most important and commonly used services on a smartphone. A recent study put web browsing as the second most common activity adults do on their smartphones [2]. However, the current web browser wastes a large amount of power during the course of a web page

load. There are two key components in use when loading a web page in a web browser on a 3G enabled phone; the CPU and the network adapter (3G or Wi-Fi).

To efficiently utilize the limited resources and balance their incurred trade-offs, 3G networks employ a resource management policy distinguishing them from wired and Wi-Fi networks [25]. In 3G networks, multiple timers are used to control the radio resource, and the timeout value for releasing the resource can be more than 10 seconds. Therefore, it is possible that the 3G network interface continues to consume a large amount of power, even when there is no network traffic [30].

Furthermore, as the complexity of web pages increases, so does the computational intensity of the web page, placing a greater load on the CPU. The CPU frequency of a smartphone is determined by that of the devices' default Linux governor; unfortunately this governor cannot always provide the optimal frequency in terms of performance and energy efficiency for each application.

In this paper, we undertake improving the power consumption of a smartphone web browser through two innovative techniques. Firstly, we target the energy consumption of the network adapters by partitioning the workflow of the web browser into 2 phases. During the course of a web page load a

number of browser specific tasks take place e.g. parsing, painting, resource loading etc. These tasks can be partitioned into 2 phases; IO bound loading and CPU bound rendering work. Our approach partitions these two so that all IO bound loading work is done first before any CPU bound rendering work can occur. This approach allows us to save energy in the network adapter by switching to a low power state earlier than that of the stock original browser.

Secondly, we focus on improving the consumption of the CPU. In order to do this, we temporarily replace the smartphones default governor with a DVFS algorithm. The DVFS algorithm exploits the partitioned architecture described above by running a low CPU frequency in the IO bound loading phase and an optimal frequency in the CPU bound rendering phase. The optimal frequency is a frequency that performs the rendering of a web page in at least the same time as the default governor, but with lower energy consumption. The calculation of the optimal frequency takes place in the cloud concurrently to the loading phase. To further unburden the resource constraints of the smartphone we use the processing power of the cloud to rapidly calculate the optimal frequency concurrently as the device is processing the loading phase. The cloud uses a combination of web page characterization and regression models to accurately predict the optimal frequency for rendering the web page on the device.

The proposed approach is implemented on an Android OS smartphone running on an emulated KT UMTS network. Experimental results show that our approach can reduce the power consumption of a web page load on a smartphone device by 12% in a Wi-Fi environment and 14.5% in a UMTS 3G network environment, with no added latency to web page load times.

The rest of this paper is organized as follows. In chapter 2 we give background information on smartphone web browsers, the WebKit rendering engine and the 3G radio interface. Chapter 3 introduces our solution overview, and in chapter 4 we present our implementation. Chapter 5 describes our experimental setup and analyses our experimental results. Chapter 6 surveys existing work in this area. Finally, we conclude with Chapter 7.

Chapter 2 - Background

In this section, we provide background information on the design of mobile web browsers, the function and design of the WebKit rendering engine, and the design and power consumption of the 3G radio interface.

2.1 - Mobile Web Browsers

In modern web browsers, the main functionality is to present the user with a web resource of their choice, by requesting it from a server and displaying it in the web browser window [16]. Commonly, a web resource is in HTML form, but can often also be CSS (Cascade Style Sheets), JavaScript, or Images files. The location of the resource is specified by the user to the web browser using a URL (Uniform Resource Locator).

The basic flow of a web browser loading a page can be broken down into 3 stages:

1. Discovery - Once the user has entered the URL into the web browser, the web browser will execute a DNS lookup to confirm the web page of the obtained URL exists.
2. Loading - fetches the requested web resources from the server or from the local cache.
3. Rendering - parses HTML documents, builds a DOM tree, parses CSS styling information and combines with information in the DOM tree to

create a render tree. Finally the web browser paints the render tree into the web browser window.

The 3 stages defined above repeat numerous times during a web page load. When opening a page, the web browser will incrementally load multiple web resources, build the DOM tree, render tree and paint to the web browser window. This iterative approach is needed for two key reasons. First, it allows for an intermediate representation of the web page to be displayed in the web browser window and adds a perception to the user that the web page is loading. Second, it often takes multiple iterations over multiple resources to finish loading a web page. This is because the web browser often discovers new resources while processing loaded ones [29].

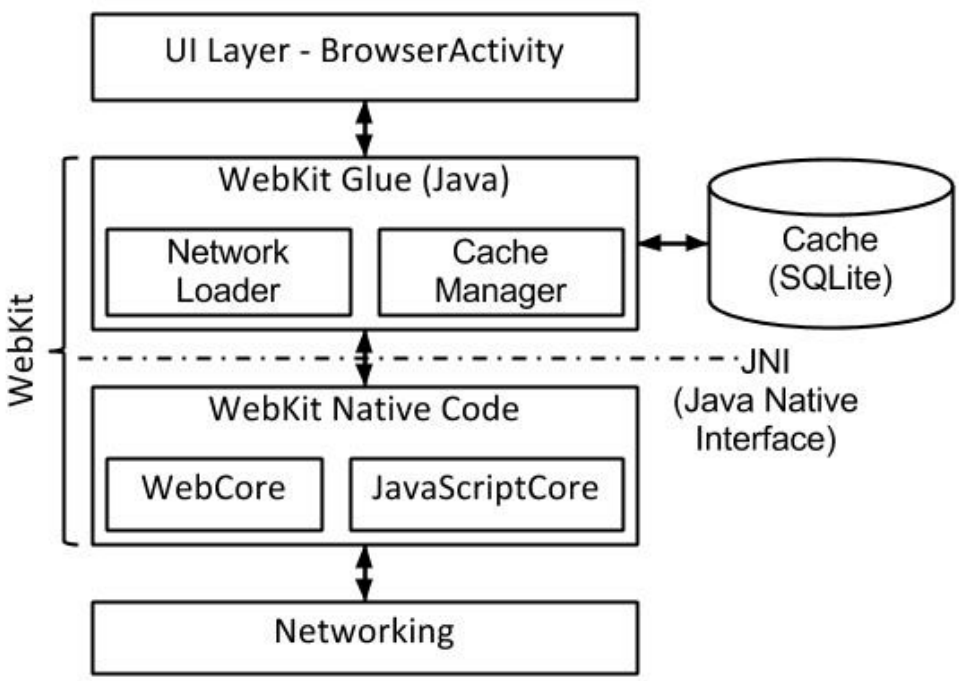


Figure 1: Default Android Web Browser Architecture

Fig. 1 shows the main components of an Android [3] web browser. The web browser consists of 7 core components

- UI Layer - all Android Activities that comprise the web browser. User interface elements like address bar, menus, forward/back buttons, and corresponding event handlers.
- WebKit Glue - marshals the actions between the UI and the rendering engine and contains the Java classes that maintain the current state of the web browser. Includes a CacheManager class which provides an interface to store and lookup from a SQL based cache. A NetworkLoader class is used to download resources from the network.
- WebKit Native Code - Contains two components: WebCore contains the native libraries which parse and renders web resources on into the web browser window. This component relies on the NetworkLoader and CacheManager class to download different components. JavaScriptCore is the component used to parse and execute the JavaScript's.
- Networking – used for network calls, like HTTP requests.
- Cache – the web browser needs to save and store a range of data to the hard disk

2.2 - WebKit Rendering Engine

WebKit is an open source rendering engine developed from a fork of KDE's HTML layout engine KHTML. As of October 2012, WebKit has the largest market share of any rendering engine at over 40% market share of all web browsers and over 90% of all mobile web browsers [8]. The WebKit rendering engine comprises the basis of many modern web browsers, including Apple iOS, Android, Apple Safari, Google Chrome and Tizen.

The main responsibility of WebKit is to render/display web resources in the web browser window, but this is not its only undertaking, it is also responsible for interacting with the network layer to download resources and also the passing of JavaScript resources to the JavaScript engine for parsing and execution. Due to this, if a working WebKit port is available for a desired architecture, a software developer can easily begin development on a new web browser by building a user interface on top of the Webkit rendering engine.

As shown in Fig.1; WebKit consists of two core components; WebKit Glue and WebKit Native code. Within WebKit Native code there exists two more components, WebCore and JavaScriptCore, the former is the core component of WebKit as it deals with parsing, layout, rendering, DOM construction and resource loading.

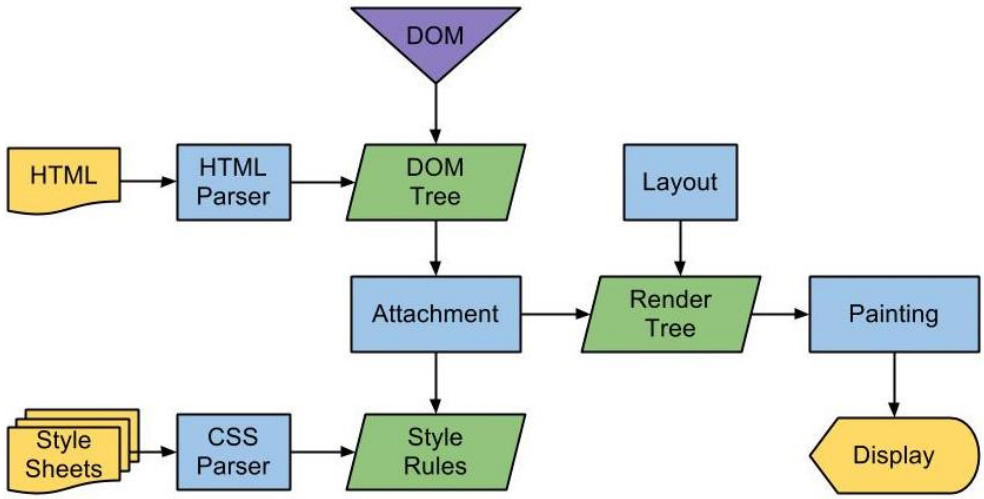


Figure 2: WebKit Main Flow

Fig. 2 shows the main flow of the WebKit rendering engine once web resources have been fetched from the network or cache [14]. WebKit starts by parsing the HTML document and turns HTML tags to DOM nodes in the DOM tree. It then parses the style data, CSS files and in style elements, to create style rules for the web page. The DOM tree, containing the content of the page, and the style rules are then combined to create the render tree. The render tree consists of panels on the webpage that can contain multiple resources with visual attributes like color and dimensions [16]. Once the render tree is constructed, a layout operation takes place, in which each node in the render tree is given precise coordinates to where it should be placed in the web browser window. Finally, the render tree is traversed and each render tree node is painted into the web browser window.

During the course of a web page load this flow is gradual and happens many times; the loading and rendering phases of the web browser are repeatedly executed and intertwined together. Often on desktop web browsers these operations can take place in parallel, resource loading can take place at the same time as parsing, layout and paint operations, but in Android this process is sequential. The implementation of WebKit within Android is single-threaded; the WebCore component runs in a single thread called WebViewCoreThread. When the default web browser is loading a page Android has two key threads WebViewCoreThread and BrowserMain thread executing. BrowserMain thread is responsible for the user interface and requesting actions from the WebViewCoreThread such as loadURL. However, WebViewCoreThread is responsible for the key operations such as HTML requests, parsing of CSS and HTML, DOM tree construction etc. As shown in Fig. 3 during a web page load, when an URL is entered by the user to the page being fully loaded, the flow of loading resources then rendering (painting) is repeated many times.

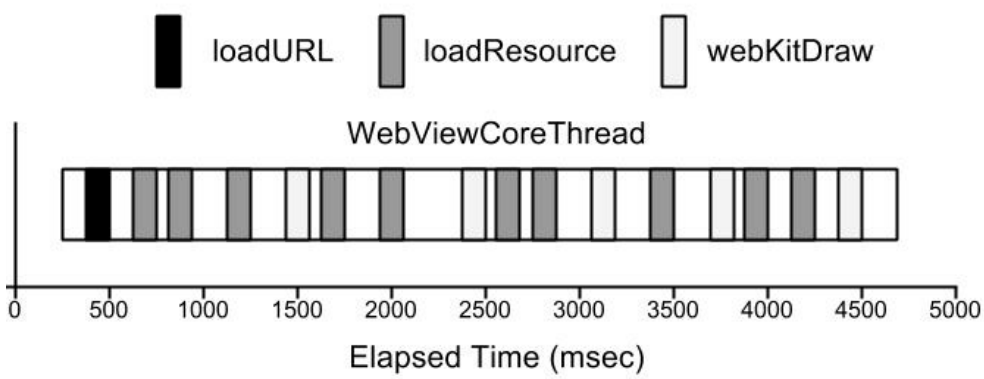


Figure 3: A Simplified Android Trace of a Web Page Load

2.3 - 3G Radio Interface

The 3G UMTS network consists of three subsystems: User Equipments (UE), UMTS Terrestrial Radio Access Network (UTRAN), and the Core Network (CN). UEs are the handsets carried by the users. The UTRAN is the radio access network, which connects the UEs to the CN. This consists of two components: base stations, which are called Node-Bs and Radio Network Controllers (RNC), which control multiple Node-Bs. The CN is the backbone of the cellular network [27].

To efficiently utilize the limited radio resources [25], the 3G radio resource control (RRC) introduces a state machine associated with each handset. A single RRC state machine is maintained at both the UE and the RNC. The two peers state machines are always synchronized though control channels except during transient and error situations [17].

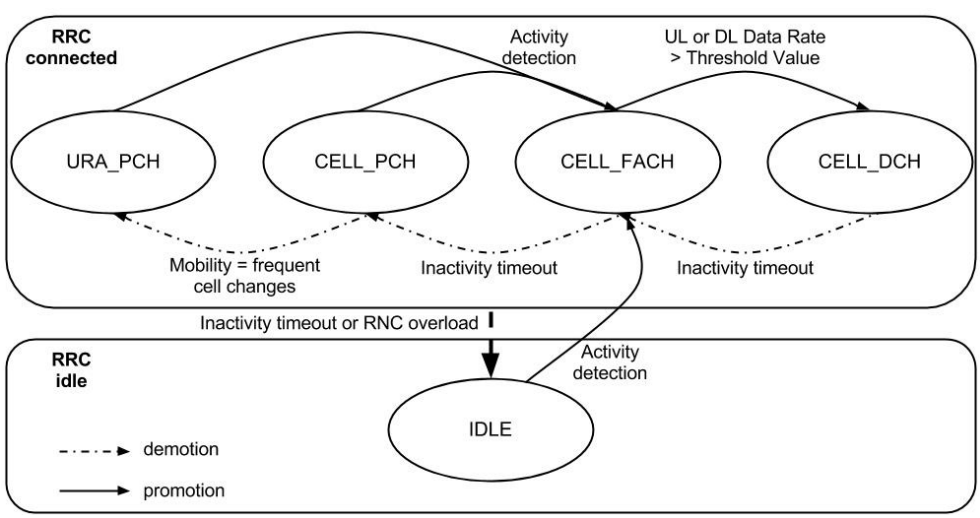


Figure 4: The 3G RRC State Machine

As shown in Fig. 4 the RRC state machine consists of two operational modes: idle and connected. When a UE is powered on it enters the idle mode; it is attached to the network but it is not engaged in data transfers. Once an RRC connection is established the UE will switch operational modes to connected mode, in this mode a UE can be in one of four states defined below [22]:

- CELL_DCH - A dedicated physical transport channel is allocated in both uplink and downlink directions to the UE. In this state the UE can fully utilize the radio resources for the transfer of user data. This state has the largest energy consumption of the RRC state machine.
- CELL_FACH - No dedicated channel is allocated to the UE. The UE can only transmit user and control data over shared transmissions channels at a relatively low speed, usually less than 15kbps. The power consumption of CELL_FACH state is roughly 50₩% of that in CELL_ DCH [26].
- CELL_PCH - The UE listens to the paging and broadcasting control channels, but uplink and data transfer is not possible. The location of the UE is known at a cell level.
- URA_PCH - This is the same as CELL_PCH in terms of functionality, but the location of the UE is known at a UTRAN Registration Area level. The power consumption of CELL_PCH and URA_PCH are close to zero.

Within the RRC state machine, there are two types of state transitions; promotions and demotions. State promotions will switch from a state with

lower radio resources and power consumption to a state which consumes more radio resources and power. State demotions on the other hand move in the opposite direction. State Promotions and demotions are dependent on the Buffer Occupancy (BO) at the Radio Link Control layer [22]. A state promotion from CELL_FACH to CELL_DCH takes place if the BO level exceeds a threshold value for a set time. All other promotions occur when any user activity is detected on the network. State demotions occur when the BO is zero and a timer of inactivity is exceeded. The transition from CELL_PCH to URA_PCH is neither a demotion nor a promotion, both states use the same radio resources and power consumption, the only difference between the states is the scope of the UE location. A UE that moves cells at a relatively slow rate will lie in the CELL_PCH state and its location will be known at a cell level. In contrast, a UE that switches cells quickly will lie in the URA_PCH state and its location will be known at a UTRAN Registration Area level. The operational mode of the RRC state machine can change from connected to idle if a timer of inactivity is triggered or the RNC is overloaded and needs to release connections.

For the rest of this paper, as we are only concerned with energy consumption of the states, I will refer to URA_PCH, CELL_PCH and IDLE states collectively as IDLE. This is due to the fact that the 3 states use approximately the same power (idle phone power).

Chapter 3 - Solution Overview

In this section, we model the target system, and introduce our strategies to improve power consumption for web browsers on smartphones. We target two components of a smartphone where we can improve power consumption in the web browser; the CPU and the 3G network adapter

3.1 - Target System

Our target system architecture is the Android framework default web browser, which uses the WebKit rendering engine. Our power saving strategies target the two components in a smartphone that account for over 40% of overall power consumed when browsing the web (excluding display) [15].

3.2 - 3G Network Adapter Energy Reduction

As the majority of 3G network adapter energy consumption takes place while the phone is not in the IDLE state, we want to redesign the architecture of the web browser in order to minimize the amount of time spent in none IDLE states (CELL_DCH and CELL_FCH).

A naive approach would be to switch the network adapter into an IDLE state as soon as the web page's last data transmission has taken place. However, this is difficult due to periodic transmissions and dynamic scheduling in HTML5 and AJAX requests. Furthermore, this approach is not optimal, as

mentioned in section 2.2, during the course of a web page load the loading and rendering phases of the web browser overlap and are repeated many times. As consequence of this approach, the 3G network adapter is frequently unable to switch into a low power IDLE state until the end or after the web page load.

In order to minimize this time spent in the CELL_DCH state and thus maximize the IDLE state time, we propose a browser architecture that partitions the loading and rendering phases of the web browser into two separate non-repeating phases. In the new architecture all data transmissions of the web page load are grouped together and performed as soon as possible. Only then, once all web page resources have been loaded with the network adapter, does the rendering of page take place. Fig. 5 shows a simplified Android trace view of the new partitioned web browser architecture.

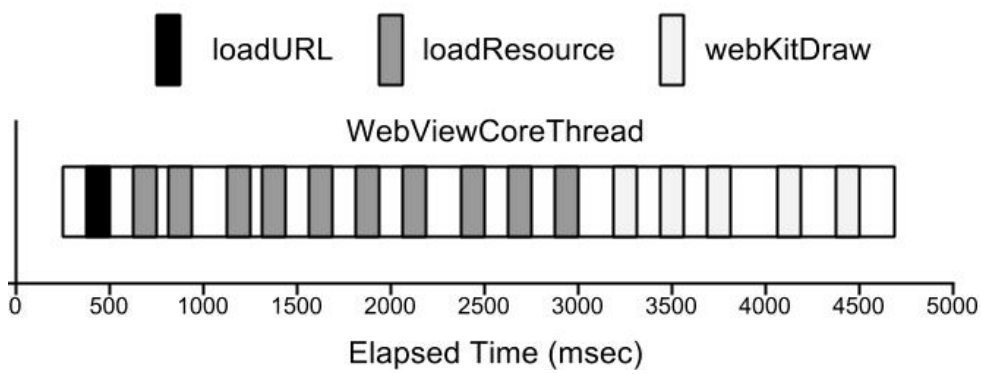


Figure 5: Proposed Partitioned Architecture Loading a Web Page

By grouping the data transmissions together at the beginning of the page load, all the resources of the web page can be downloaded in a shorter time, allowing us to switch the phone into an IDLE state (via the CELL_FACH state) faster than in the original architecture. Hence, we can save 3G network adapter energy, as less time is spent in the higher energy consuming CELL_DCH and CELL_FACH states.

3.3 - CPU Energy Reduction

By partitioning the web browser into two non-repeating loading and rendering phases we effectively partition the workload of the web browser into an I/O bound phase during loading and a CPU bound phase during rendering. By using a DVFS algorithm on the partitioned browser we can effectively exploit the different bound phases and significantly reduce the power consumption of the CPU while still keeping the web page load time within the bounds of the original browser.

3.3.1 Loading Phase

Three tasks take place during the I/O bound loading phase of the partitioned browser:

1. Parsing – involves parsing of both CSS style sheets and HTML
2. Resource Loading – fetches a resource given its URL, either from the remote web server or from local cache

3. DOM tree construction – the document object model tree is created with the document at its root and elements are added to the tree [29]

Even though we effectively call this loading phases I/O bound, the first and third tasks of the loading phase are actually CPU bound events. Even so, these tasks they are not heavily CPU intensive. Due to this, the loading phase of the web browser is relatively undemanding on the CPU and thus allows us to effectively set the CPU frequency low during this phase.

3.3.2 Rendering Phase

The rendering phase of the partitioned web browser is an intensive CPU bound phase of a web-page load. During the course of the rendering phase the following tasks take place:

1. Layout – computes and updates the screen locations and size of the DOM elements
2. Render Tree Construction – applies style rules and layout coordinates to DOM elements to create the a render tree in order of elements to be displayed
3. Painting – render tree is traversed and its elements and rendered onto the display. Generates the final graphic representation of the web page

On the basis of the fact that different webpages take different rendering times and use different amounts of energy based on their individual characteristics

[31], we propose a DVFS approach that aims to find an optimal frequency for rendering a specific page based on its characteristics. The optimal frequency is a CPU frequency that can render the web page in at least the same time as the default ONDEMAND governor, while consuming less energy.

Fig. 6 shows the energy consumption and rendering time of the MSN web page at the 5 available frequencies and the ONDEMAND governor on the Samsung Galaxy S2. Higher frequencies can render the web page faster than the ONDEMAND governor, but often consume more energy. Lower frequencies exhibit the reverse behaviour, slower rendering time and less energy consumption. However, there is often an optimal point or "sweet spot" that refers to a frequency that is both faster and consumes less energy when rendering pages compared to the ONDEMAND governor. As we can see from fig. 6, the optimal frequency for the MSN web page is 800Mhz.

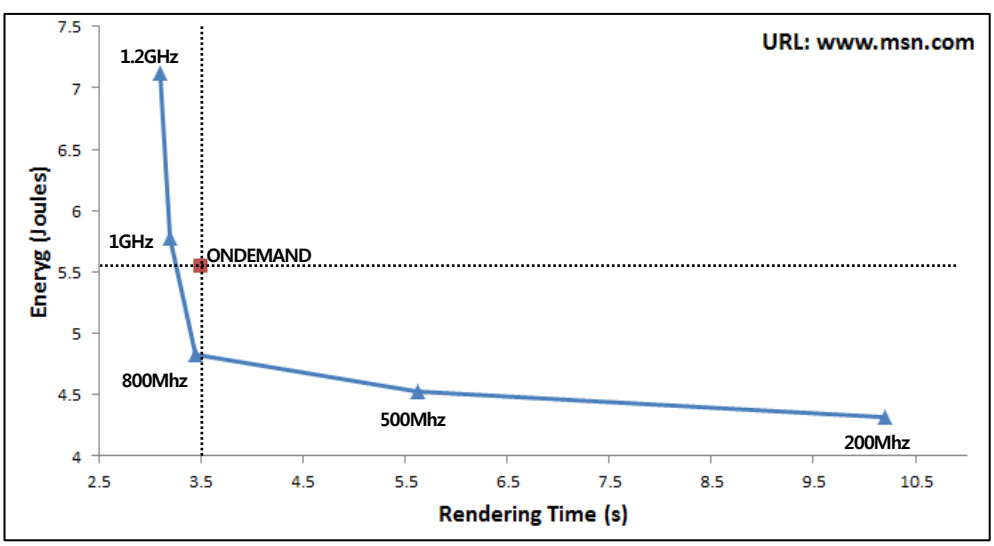


Figure 6: Optimal Frequency Graph for URL http://www.msn.com

Unfortunately as web page load times and energy consumption varies based on their characteristics, there is not always an optimal frequency compared with the ONDEMAND governor. Fig. 7 shows the optimal frequency graph for the BBC web page, in this case an optimal frequency does not exist, and the ONDEMAND governor outperforms the alternative frequencies. In this situation the DVFS algorithm would simply run the rendering phase with the ONDEMAND governor.

In order to find an optimal frequency we need the ability to predict the rendering time and energy consumption of a web page. To do this, we needed to extract key characteristics of a web page, and feed them into a number of prediction models to estimate the rendering load time and energy consumption at different frequencies.

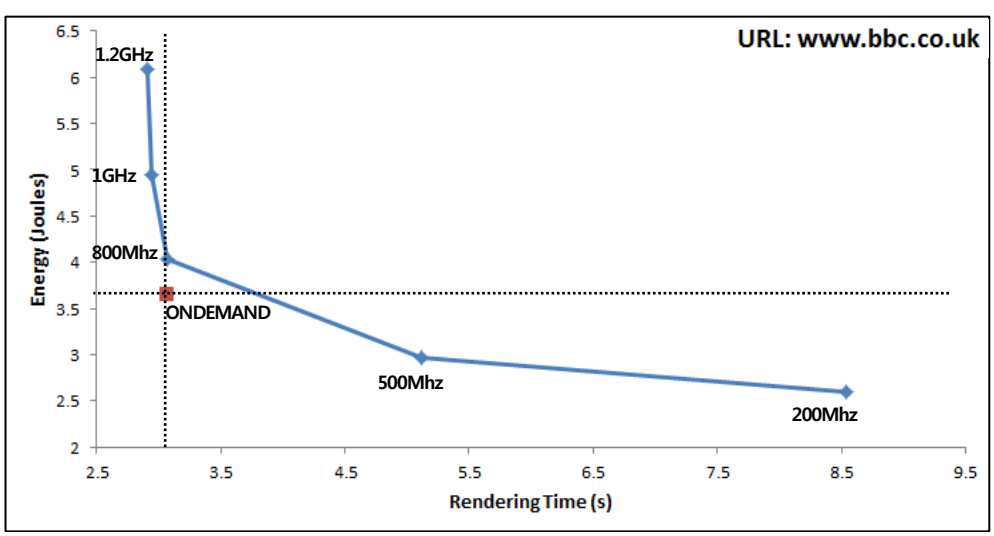


Figure 7: Optimal Frequency Graph for URL: www.bbc.co.uk

For our prediction models we used a regression model that captured different aspects of web page characteristics and their interactions thus allowing us to predict web page rendering time and energy consumption [31].

A regression model is a statistical technique for estimating the relationships between a set of predictors and a response. In this case, the response from the regression model will be the load time or the energy consumption of rendering the web page and the predictors are a set of web page characteristics. As the root cause of web page variance arises from the inherent "web page variance" in the structural (HTML) and style (CSS) information [31], we mined web pages from the Alexa top 50 sites [1] and extract key HTML and CSS attributes to be used as the models predictors. In addition to this, we also required further sampling observations to train the model, which are rendering load times and energy consumptions.

The regression model used in this paper is a linear regression model which models the web page's rendering load time and energy consumption as a linear combination of various web page characteristics (predictors), calculated by: $\mathrm{y} = \beta_0 + \sum_{i=1}^{p} x_i \beta_i$ where y denotes the response, $x = x_1, \ldots, x_i$ denote the p predictors, and $\beta = \beta_0, \ldots, \beta_p$ denote the corresponding coefficients of each predictor.

Once the regression models are successfully trained they can then be used to calculate the predicted rendering time and energy consumption and thus the

optimal frequency of a specific web page during the loading phase of the web browser. As the process of calculating this optimal frequency, extracting web page characteristics and estimation with prediction models, would add to the CPU overhead and energy consumption in the loading phase of the web browser, we have moved these tasks to the cloud.

Fig. 8 depicts the new partitioned architecture of the web browser with the DVFS algorithm. The web browser is partitioned into two non-repeating phases, loading and rendering, to reduce energy consumption in the 3G network adapter. The loading phase of the web browser is an I/O bound workload and runs at a low CPU frequency. The rendering phase of the web browser runs at an optimal CPU frequency calculated by a cloud server during the loading phase. When a web page URL is loaded in the browser, it is not only passed to the rendering engine on the smartphone device, but also to the cloud server to find the optimal frequency to render the web page. Within the cloud server, key HTML and CSS components are extracted from the web page to characterize its workload. The characterized web page is then fed into a number of prediction regression models that estimate the rendering time and energy consumption of the device and its available CPU frequencies. The estimated times and energy values are then compared to the ONDEMAND governor values and an optimal frequency is chosen and returned to the phone. If no available frequency outperforms the ONDEMAND governor, the ONDEMAND governor is set as the frequency.

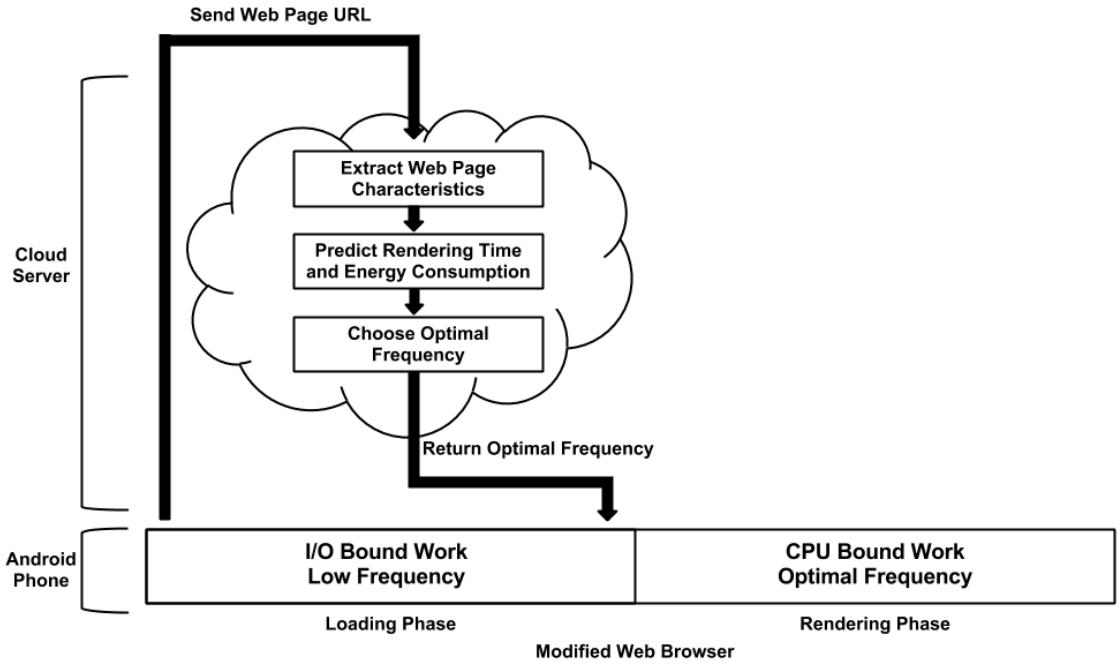


Figure 8: Modified Partitioned Web Browser Architecture with Cloud-Assisted DVFS Algorithm

Chapter 4 - Implementation

In this section we present the proposed DVFS energy-aware web browser in detail. The implementation of the proposed scheme is divided into three stages; partitioning the workflow of the web browser, implementing a DVFS algorithm and calculation of an optimal rendering CPU frequency. We implemented the web browser by modifying the stock Android web browser and underlying WebKit rendering engine on a Samsung Galaxy S2 I9100 device running CyanogenMod 9.1.0 [5]. CyanogenMod 9.1.0 is an aftermarket firmware for smartphones based on the Android 4.0.4 Ice Cream Sandwich operating system.

4.1 – Partitioned Web Browser

To partition the workflow of the web browser we need to modify the underlying WebKit rendering engine of the stock Android web browser. SuppressIncrementalRendering is an option in the current WebKit source tree that disables the natural incremental rendering of the rendering engine and only renders content once it is fully loaded into memory [13]. To mask the effects of incremental rendering it ensures all sub-resources have loaded and a full layout has taken place before painting the document for the first time. This functionality is achieved by exploiting the ready state machine of the DOM

tree. The DOM tree ready state machine in WebKit consists of the following 3 states:

- Loading – the document is loading
- Interactive – the document has been parsed and is loading sub-resources
- Complete – the document has been parsed and all resources/sub-resources have been loaded from the network or cache

If the SuppressIncrementalRendering option is enabled within the rendering engine, we suppress paint and composite layer events until the DOM tree is in the complete ready state. As all resources and sub resources are loaded from the network or cache first before any painting occurs, enabling this option would effectively partition the workflow of the browser into 2 non-repeating phases; loading and rendering.

Unfortunately, in the version of WebKit in Android 4.0.4, 534.30, this option is not present. To achieve this desired behaviour we ported the functionality from the current WebKit source tree to the Android WebKit source. To port this option to the Android WebKit source we modified the following files in the WebCore component of WebKit:

- Settings – contains the settings of the browser e.g. plugins enabled, java disabled etc.
    - Modifications - adding a Boolean variable to

"SuppressIncrementalRendering".

- Document – manages and maintains the DOM tree.
    - Modifications – If SuppressIncrementalRendering is enabled in settings, only allow rendering to occur once the DOM tree is in its "complete" ready state.
- Frameview – represents the containing view for the document and is responsible for managing the layout of the rendering tree.
    - Modifications – methods added to render the whole tree from the root node
- RenderLayer – responsible for painting layers onto the screen.
    - Modifications – modify the paint methods to check if suppressIncrementalRendering is enabled, and if so to simply return from the method before any painting can occur.
- RenderLayerCompositor – responsible for painting layers on top of other layers (compositing).
    - Modifications - modify the paint methods to check if suppressIncrementalRendering is enabled, and if so to simply return from the method before any painting can occur.

The suppressIncrementalRendering option was intended to be used by web application developers to make web views look more like native views, as the whole document would be loaded in one shot. To our knowledge, this is the first time an approach has been proposed that uses this option to save web browser energy on a 3G smartphone device.

4.2 – DVFS Algorithm

In the Linux kernel under the Android operating system, the CPUfreq subsystem provides a modularized interface to manage the CPU frequencies. The policy manager for power management is called a Governor in Linux, which controls the CPU frequency through the interface of CPUfreq [19]. Several kernel-level governors are supported by Linux for CPU frequency management. The default governor for DVFS management within many Android devices is the ONDEMAND governor. The ONDEMAND governor is a dynamic governor that switches to the maximum clock frequency when there is a load on the CPU and then, gradually steps back down through the CPU frequencies when the CPU load abates. Linux also provides a Userspace governor, which allows userspace programs to set the CPU's operating frequency through the Linux sysfs interface. In order to change the governor, governor settings and CPU frequency, Android provides the following special files on the file system of the smartphone device (within /sys/devices/system/cpu/cpu0/cpufreq):

- scaling_cur_freq - contains the current CPU frequency
- scaling_governor – contains the current governor
- scaling_max_freq – contains the maximum frequency the CPU is allowed to run at

- scaling_min_freq – contains the minimum frequency the CPU is allowed to run at
- scaling_setspeed – writing to this directory sets the current speed of the CPU

To change the CPU frequency of the Android device, first we must change the current governor to userspace by writing to the scaling _governor special file. Then we write the desired frequency into the scaling_setspeed special file. The Samsung Galaxy S2 allows us to change the CPU frequency to any of the following speeds: 200Mhz, 500Mhz, 800Mhz, 1Ghz and 1.2Ghz.

The DVFS algorithm sets the frequency of the CPU at 2 points during the course of the web page load:

1. The beginning of the loading phase – sets CPU frequency to 500Mhz
2. The beginning of the rendering phase – sets CPU frequency to optimal

To ensure the CPU frequency is changed as early as possible within each stage, we set the CPU frequency at the earliest possible method in the source code of the browser or the rendering engine.

As soon as a URL is requested by a user, a loadURL() method is called within the browser application code in android (tab.java), this is the point where we set the governor to userspace and set the CPU frequency to 500Mhz:.

The clock frequency is set to the optimal value when all resources and sub resources have been loaded from the network or cache, this is signified by the didFirstLayout() method being called from within the WebKit native glue engine (BrowserFrame.java).

4.3 – Optimal Rendering CPU Frequency

The calculation of an optimal CPU frequency is divided into 2 stages; firstly, implementation of a web page characterizer and secondly, training of regression models to calculate web page rendering time and energy consumptions for each available frequency.

The web page characterizer is a web page profiler that extracts key information about a web page. The web page profiler is written in Ruby and takes a URL as its argument and returns a list of characteristics for the web page. Fig. 9 shows a table of the key characteristics that are returned from the web page characterizer. The HTML tag characteristics are extracted using the Nokogiri HTML parser ruby gem [10]. Only a few HTML tags types are hot

Catergory	Characteristics
HTML Characteristics	Number of dom tree nodes
	Number of references
	Number of div tags
	Number of li tags
	Number of span tags
	Number of table tags
	Number of p tags
CSS Characteristics	Number of selectors
	Number of id selectors
	Number of class selectors
	Number of descendent selectors
	Number of psuedo selectors
	Number of color properties
	Number of display properties
	Number of width properties
	Number of float properties
Web Page Content	Total Size of All Images

Figure 9: Extracted Web Page Characteristics

across all web pages, so we only extract the total number of the top 6 hottest used tags. We extract the number of DOM tree nodes as this is a strong heuristic indication of a web page's processing complexity. CSS characteristics are extracted from the css scripts using the CSS-parser ruby gem [12]. The number of CSS selectors is also a strong indication of the computational complexity of a web page as each rule has to be checked against each HTML tag. Thus giving a computational complexity in the order of O(# of tags X # of rules). We also extra the total number of the top 4 hottest used CSS selector patterns and CSS properties [31].

As mentioned in section 3.3.2, we use a linear regression model which takes the list of web page characteristics as predictors and responds with either a predicted web page rendering time or energy consumption. The response is calculated by: $y = \beta_0 + \sum_{i=1}^{p} x_i \beta_i$ where y denotes the response, $x = x_1, \ldots, x_i$ denote the p predictors, and $\beta = \beta_0, \ldots, \beta_p$ denote the corresponding coefficients of each predictor.

We create 2 models for each available CPU frequency on the smartphone device. One of each pair predicts the total rendering energy consumption, while the other predicts the total rendering time. In the case of the Samsung Galaxy S2, we create 6 pairs of models, 5 pairs for the 5 available frequencies in the userspace governor and 1 pair for the ONDEMAND governor.

To construct the regression models we need to use a number of sampling observations to find the best fitting coefficients β. We take 25 of the top 50 Alexa web sites and put them through our web page characterizer. We then measure the rendering time and energy consumption of each page at each available frequency using the experimental setup described in section 6. We download the pages to the device and load them locally to ensure our energy and time measurements are only for the rendering phase and do not include any network loading work. Fig. 10 shows the 25 sample pages and their

Web Pages	Dom Tree Nodes	CSS Characteristics								HTML Characteristics								Total Image Size
		Selector Patterns				Properties				Tags Count	Tag Types							
		ID	Class	Descendant	Pseudo	Color	Display	Width	Float		Links	Div Tags	li Tags	img Tags	Span Tags	table Tags	p Tags	
www.msn.com	2483	175	2243	2320	456	567	329	252	346	1369	316	438	227	25	84	9	16	160760
seoul.craigslist.co.kr	778	63	556	625	70	57	158	118	49	725	301	16	271	0	0	34	1	0
www.huffingtonpost.com	20407	2688	16578	10685	1521	9968	887	3088	1169	3595	1018	837	622	154	440	0	137	2534314
www.ask.com	245	16	148	129	11	2	10	77	54	294	53	174	0	12	22	0	0	730442
www.weather.com	3948	413	3215	2724	414	766	301	877	246	711	182	148	141	23	42	0	3	3957937
www.foxnews.com	1293	680	583	1152	144	235	168	340	207	1240	325	146	240	78	38	0	119	162591
www.nytimes.com	2902	1127	1505	2044	184	436	342	598	332	1492	386	253	238	62	42	12	43	476017
www.apple.com	3614	1691	759	1810	817	328	622	702	362	136	28	15	21	10	10	0	2	383564
www.adobe.com	41738	2488	34476	13575	7354	9460	3246	6575	3990	908	288	189	196	13	36	10	43	99901
www.walmart.com	5939	497	5183	2957	147	800	516	1504	464	1200	576	273	89	44	28	20	8	374155
www.yelp.com	6008	1019	4625	3287	289	680	673	1464	535	1399	325	276	315	50	137	0	19	126877
www.ehow.com	1723	342	1141	953	160	375	260	206	89	489	138	25	91	22	12	0	11	294410
www.guardian.co.uk	6663	1733	3061	4886	668	3087	397	704	498	1781	541	288	324	103	96	0	98	800065
www.gumtree.com	1404	391	710	880	165	185	253	222	97	1017	370	39	356	32	42	0	4	6036
www.rightmove.co.uk	3593	1309	1302	2213	443	848	396	794	473	349	113	48	91	12	7	0	10	24329
www.tesco.com	40	2	37	36	5	0	0	0	0	822	187	90	134	0	104	0	69	1297603
www.dailymotion.com	1907	208	943	1122	320	414	201	349	191	583	100	188	46	20	150	0	0	317956
www.stackoverflow.com	1839	235	1400	1094	150	479	177	403	170	2349	640	1300	52	17	193	7	0	8737
www.cnet.com	6720	3314	3056	4987	1555	1771	1527	1569	1037	1081	305	157	198	70	78	0	36	483440
www.naver.com	2362	224	2070	1864	199	478	296	823	121	851	193	107	95	45	154	0	13	198494
www.dailymail.co.uk	31021	1065	28926	27577	4434	7521	3071	4367	2090	5460	1004	973	703	455	1369	9	180	4247709
www.nbcnews.com	22218	6474	13246	15845	2469	5072	2006	3055	950	2424	577	901	219	59	119	1	83	311083
www.taobao.com	1677	506	1026	1259	138	354	218	431	220	2800	1143	245	335	122	279	0	14	1260218
www.sky.com	6302	3849	2045	4971	711	766	1233	1128	385	1505	422	439	127	27	174	0	32	573148

Figure 10: Sample Observation Web Pages and their Extracted Characteristics

various extracted characteristics.

We use R, a language and an environment for statistical computing and graphics, and its lm package to construct and tune our linear regression models. We tune our models by eliminating insignificant predictors.

To calculate the optimal frequency we extract the desired web pages' characteristics using the web page characterizer then feed the extracted information through the regression models to produce a list of predicted times and energy consumptions for each frequency. We then set the optimal frequency value based on the following algorithm:

```
Temp_optimal = 'ondemand'
if ((ondemand_time) >= predicted_1.2Ghz_time) {
        if ((ondemand_energy) > predicted_1.2Ghz_energy) {
                Temp_optimal = '1200000'
        }
}
if ((ondemand_time) >= predicted_1Ghz_time) {
        if ((ondemand_energy*) > predicted_1Ghz_energy) {
                Temp_optimal = '1000000'
        }
}
if ((ondemand_time) >= predicted_800Mhz_time) {
        if ((ondemand_energy) > predicted_800Mhz_energy) {
                Temp_optimal = '800000'
        }
}
if ((ondemand_time) >= predicted_500Ghz_time) {
        if ((ondemand_energy) > predicted_500Mhz_energy) {
                Temp_optimal = '500000'
        }
}
if ((ondemand_time) >= predicted_200Ghz_time) {
        if ((ondemand_energy) > predicted_200Mhz_energy) {
                Temp_optimal = '200000'
        }
}
Optimal = Temp_optimal
return Optimal
```

This ensures that the optimal frequency at least takes the same rendering time or less than the default ONDEMAND governor and consumes less energy.

We host the 2 scripts, web page characterizer and optimal frequency calculations based on regression models, on an apache HTTP server. We create a PHP script that can be called remotely from a smartphone device to join the two scripts. The PHP script is passed a URL and calls the 2 other scripts before returning the optimal frequency.

The modified web browser calls the PHP server script as soon as a URL is requested by the user, within the loadURL() method of the web browser package. The PHP script request is forked as a new thread to ensure the call does not block the web page load.

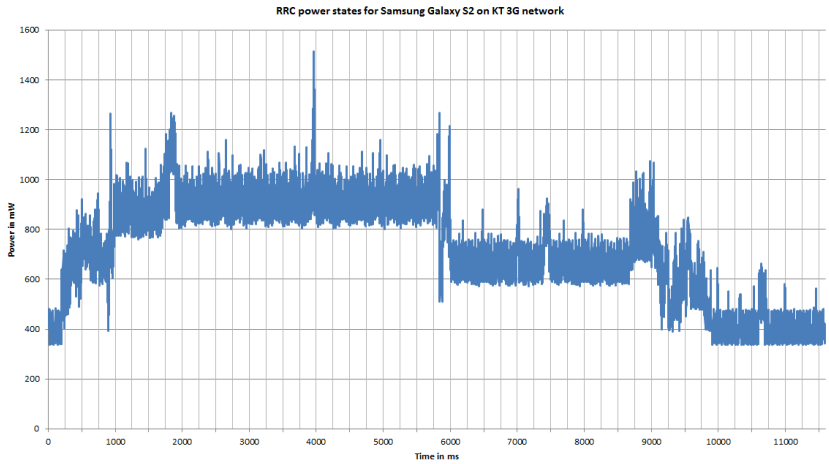
Chapter 5 – Experimental Evaluation

In this section we introduce the experimental environment and the evaluation results are presented.

5.1 – Experimental Setup

Experimental Equipment: We use a Samsung Galaxy S2 with CyanogenMod 9.1.0 OS version (equivalent to Android 4.0.4 ICS) connected to a KT 3G UMTS network as our testbed. To accurately measure the power consumption of the smartphone device we use the Monsoon Power Monitor [7]. The Power Monitor provides the current to the phone with a constant voltage of 3.7v instead of using the battery and samples at a 5KHz (200us) sampling frequency.

Network Conditions: We measure the web page load time and energy consumption under two types of network: Wi-Fi network and emulated 3G UMTS network. To emulate the 3G network we use AT&T's ARO (application resource optimizer) [4], which can accurately estimate radio energy consumption by using the inferred RRC states [23]. Fig. 11 shows the average power consumption for each RRC state and RRC promotion and tail timers of a Samsung Galaxy S2 running on the Korean Telecom 3G UMTS network. This power model along with accurate packet traces from the ARO data



RRC State Power	
P(IDLE)	450mW
P(FACH)	650mW
P(DCH)	950mW
P(FACH->DCH)	800mW
P(IDLE->DCH)	650mW

Tail Timers and Promotion Delay	
DCH->FACH timer	5 seconds
FACH->IDLE timer	3 seconds
IDLE->DCH promotion	1.5 seconds
FACH->DCH promotion	1 second

Figure 11: Measured Average Radio Consumption and Tail Times of Samsung Galaxy S2 on KT 3G UMTS Network

analyser allows us to accurately quantify the resource utilization of the original and modified web browsers. ARO can provide us with accurate profiles of device radio energy consumption, DCH occupancy time (a measure of radio resource utilization), and total state promotion times (signalling overhead).

Benchmark Web Pages: We take 5 of the top 50 Alexa web pages as our benchmark, where each web site varies in structure and use. Most web pages have 2 versions, a desktop and mobile version; the scope of this paper only considers the desktop version of the web pages as not all web pages have a

mobile version. Web pages are downloaded and hosted on an Apache HTTP server, to ensure consistency throughout experimental results.

Web Page Load Time: To accurately measure the full load time of a web page we inserted measurement timers into the source code of the browser. As soon as a user loads a URL (loadURL() method) the timer begins, only once the web page has fully completed (onPageFinished()) the timer stops and is written to a log file on the phones file system.

JavaScript and System Environment: The phones system environment was kept at a constant for all experiments: display brightness was set at a minimum; screen rotation was turned off, screen sleep off and lock screen off. In addition to this, we disabled JavaScript within the browser. JavaScript is out of the scope of this paper, as we believe that the execution of JavaScript is a related, yet separate problem from the web page load. Google recommends in its best practices for optimizing the mobile web to defer the parsing of JavaScript until after the page has loaded [11].

The variance in web page load times and energy consumption between runs in negligible. Therefore, we measure the energy consumption and web page load time across 10 runs for each experiment and average the results. Furthermore, between tests all browser caches are cleared. We will test 4 types of web browser in all tests:

1. Stock Android Browser – Stock
2. Partitioned Browser with no DVFS - Partitioned
3. Partitioned Browser with loading phase DVFS – Half DVFS
4. Partitioned Browser with full DVFS algorithm – Full DVFS

5.2 – Wi-Fi Network Environment

We compare the 4 web browsers loading the 5 testbed web pages in a Wi-Fi environment. Fig. 12 shows the energy consumption in each browser of the 5 benchmark web pages running in a Wi-Fi environment. All modified web browsers, consume less energy than the stock android browser, with the Full

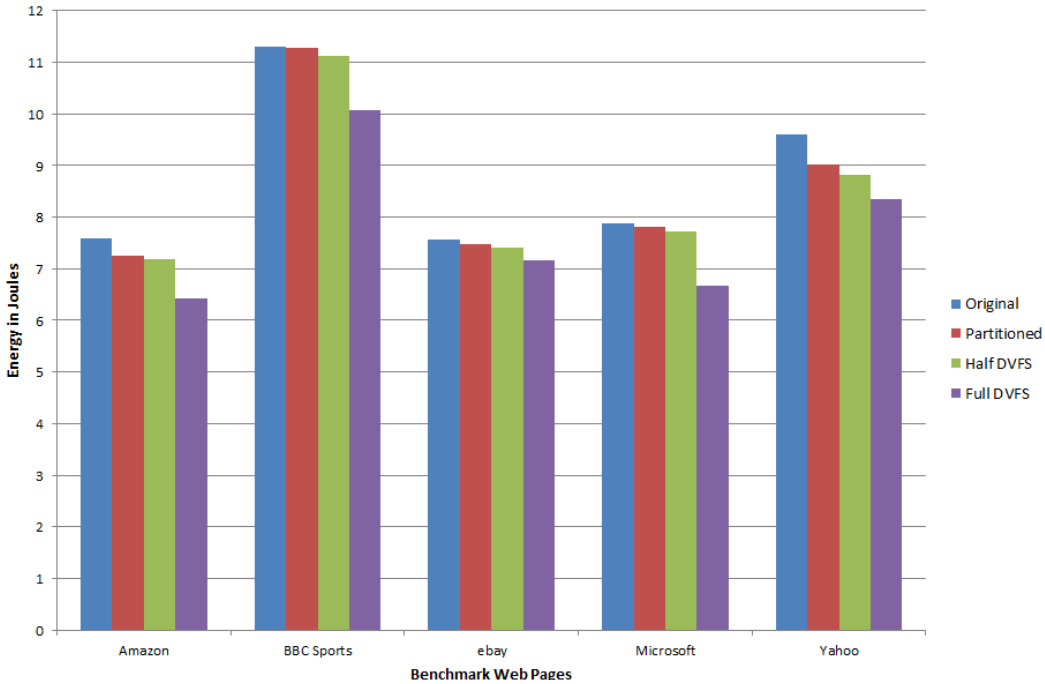


Figure 12: Energy Consumption of Benchmark Web Pages in Wi-Fi Network Environment

DVFS implementation achieving the least energy consumption across all benchmark web pages; an averaged 12% energy reduction in web page loads compared with the stock browser. The Half DVFS web browser on average consumes 4% less energy than the stock web browser. This suggests that the use of an optimal frequency in the rendering phase achieves, on average, an extra 8% reduction in energy consumption over rendering with the ONDEMAND governor. While, performing loading at a low frequency can save an extra 1.5%. Interestingly, even the partitioned browser marginally outperforms the stock browser, average reduction of 2.5%, despite being designed to save energy in 3G network environments. This is feasibly due to

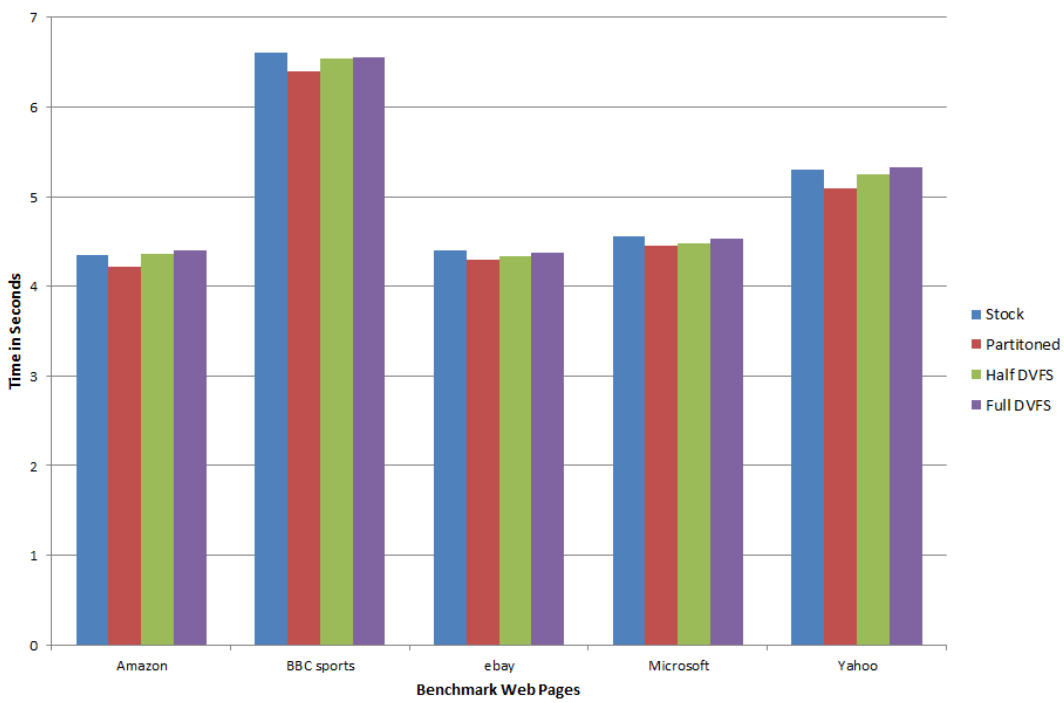


Figure 13: Web Page Load Time of Benchmark Web Pages in Wi-Fi Environment

the shorter time the Wi-Fi network adapter is being used for during a page load compared to the stock browser. The difference between the partitioned and Full DVFS browsers gives us the energy reduction of the CPU on account of the DVFS algorithm. The DVFS algorithm saves the CPU on average 9.5% during web page load.

Fig. 13 displays the web page load times across the benchmark web pages in the Wi-Fi environment. Significantly, none of the modified browsers are slower than the stock web browsers. The partitioned web browser loads the benchmark web pages fastest, on average 3.1% faster than the stock browser. We estimate this is as the partitioned web browser downloads the resources of the web page faster. The two DVFS web browsers and the stock web browser web page load times are comparable. The difference between the partitioned browser and the Half DVFS browser signifies the overhead in switching frequencies in userspace. The overhead to switch frequency at the beginning of the loading and rendering phases in the Half DVFS browser is on average 0.042 seconds. The difference between the Full DVFS browser and Half DVFS browser signifies the overhead of forking a new thread to call the PHP script to calculate the optimal frequency; this is on average 0.045 seconds.

5.3 – Emulated 3G Network Environment

To emulate the 3G network environment we loaded the benchmark web pages

in a constrained Wi-Fi environment. To mimic the bandwidth and environment of the 3G network the Wi-Fi network was conditioned using Apple's Network Link Conditioner [9]. The Network Link Conditioner allows us to limit the uplink and downlink bandwidth, delay and jitter on the Wi-Fi network to imitate that of an average case 3G network. We loaded each benchmark web page 10 times while running the ARO data collector in the background of the phone. The ARO data collector collects all packets traces in both directions. We then loaded the trace information and power model information described in Fig. 11 to analysis the web browsers resource utilization of a Samsung Galaxy S2 in an emulated KT 3G UMTS network.

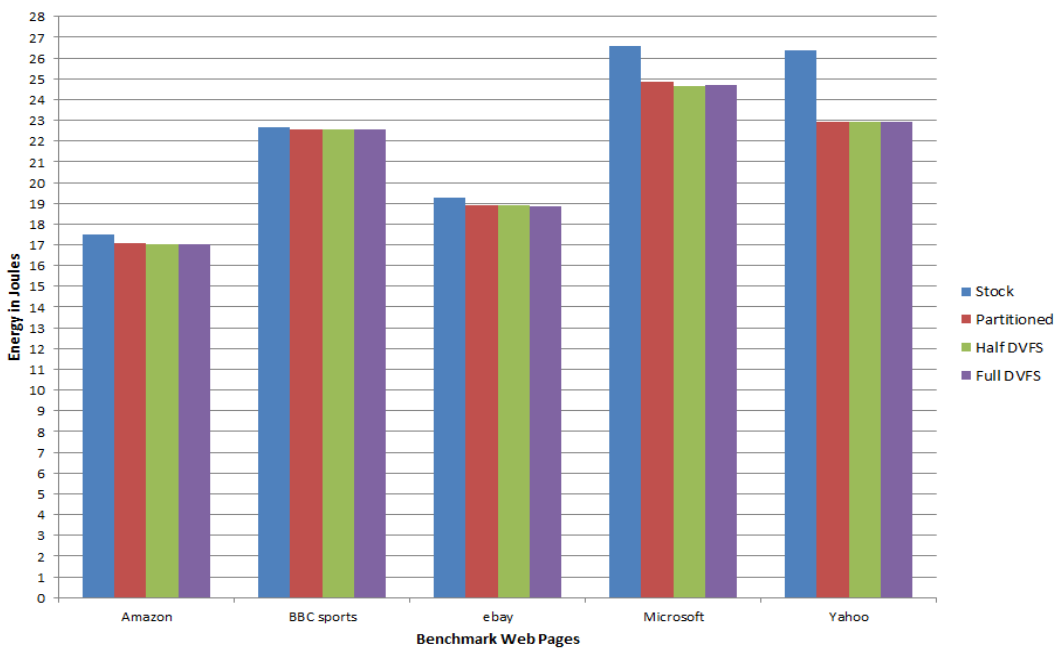


Figure 14: RRC Loading Energy of Benchmark Web Pages in the Emulated 3G Environment

Fig. 14 shows the RRC loading energy of the benchmark web pages in emulated 3G environment. RRC loading energy is the energy consumed by the RRC state machine while the web page is loading. All 3 modified web browsers consume less RRC state energy than the stock browser. On average the modified partitioned browsers consume 5% less energy in the RRC state machine than the stock browser. As expected the 3 modified browsers RRC energy consumption are comparable, as they all have the same underlying partitioned workflow. By combining the average energy reduction of solely the DVFS algorithm from the Wi-Fi experiment (9.5%) with the average reduction from the partitioned browser in the emulated 3G environment (5%), we can calculate the combined energy reduction of both the CPU and 3G network adapter in a 3G network environment. Thus, in a 3G environment the partitioned browser with cloud-assisted DVFS can on average reduce energy consumption by at least 14.5%

Fig. 15 shows the time each benchmark web page spent in the highest power DCH RRC state. The less time spent in this state signifies a better network resource utilization. As with the RRC state energy consumption, all 3 modified browsers spend less time in the DCH state compared with the stock browser. This indicates that the partitioned workflow possess improved 3G radio resource utilization in comparison to the stock browser. The partitioned architecture on average utilizes the 3G radio resource 9.2% greater than that of

the stock workflow. This indicates that an adoption of this partitioned workflow on all devices on a network could potentially reduce the traffic load on a network by 9.2%.

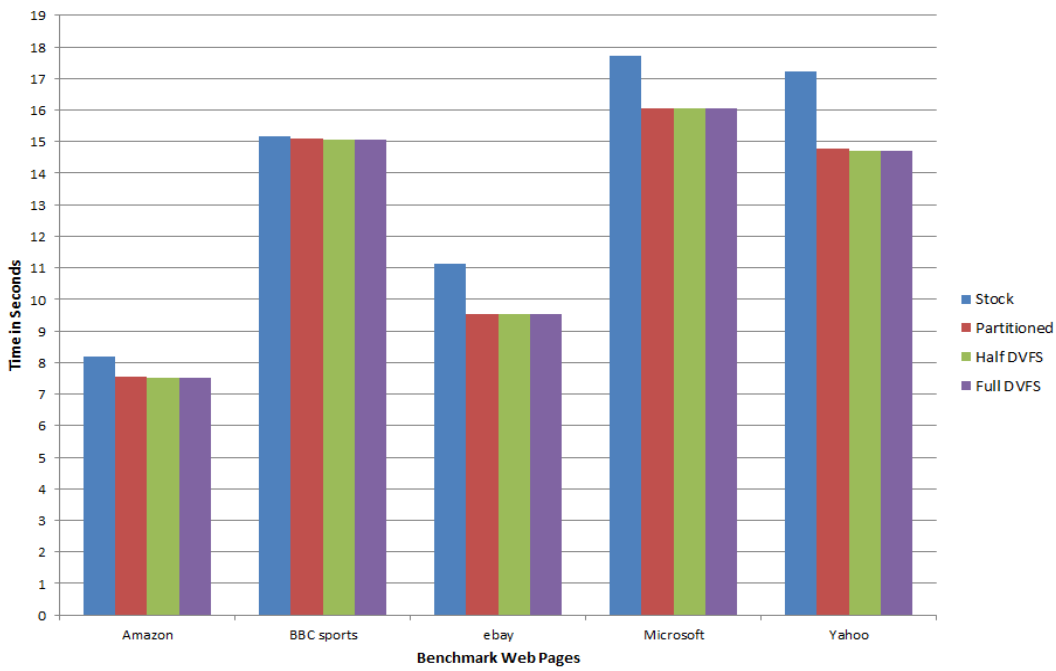


Figure 15: Time Spent in DCH State when loading Benchmark Web Pages in the Emulated 3G Environment

Chapter 6 – Related Work

There is a substantial amount of work that focuses on the energy consumption and network activity of smartphone devices. Much research focuses on ways to analysis the energy of different components in smartphones and provides performance optimizations based on this analysis.

Analysis of Energy Consumption in Smartphones: Pathak et al [21] presented advances towards understanding and automatically detecting software energy bugs on smartphones. Thiagarajan et al [28] presented an infrastructure for measuring the precise energy consumption of rendering a web page, as well as introducing tools to measure the energy consumption to render individual web elements and give recommendations on how to design energy-aware web pages for smartphones. Qian et al [24] focused on inefficiencies in web caching on smartphones. Their findings show that 18% of total HTTP traffic volume is accounted for by redundant transfers due to web caches not fully supporting protocol specifications.

Energy Saving DVFS Techniques: Carroll and Heiser [15] broke down the analysis of power consumption by a device's main hardware components. Furthermore, they showed that DVFS can significantly reduce the power consumption of the CPU. However, they report that DVFS can also increase energy consumption; as the runtime of the workload can also increase. Liang

and Lai [19] further contribute to research into DVFS mechanisms on smartphones by developing and implementing a critical speed-based DVFS mechanism for the Android operating system. Their research refers to a critical speed, which is a CPU speed that can decrease the energy consumption of an application.

Web Browser Energy Consumption Optimizations: Jones et al [18] research proposal focuses on parallelizing specific tasks of a web browser to become sufficiently responsive and energy efficient. They target parallelizing browser tasks such as parsing page layout, and scripting. Zhao et al [30] examines how to decrease power consumption for web browsing by targeting the 3G RRC state machine. They propose two techniques to save power consumption in the DCH and FACH state of the RRC state machine. Firstly, they reorganize the computation sequence of the web browser, switching the 3G radio interface into the low power idle state as soon as all data transmissions have complete. Secondly, they propose a data mining technique to predict the user reading time of a web page. Allowing the smartphone to switch to a lower power state when the reading time of a page is longer than a threshold. Zhu and Reddi [31] investigate the trade-off between high-performance and energy efficiency of switching frequencies on big/little cores. They specifically target the trade-off in scheduling the rendering of web pages Furthermore, they show that predictive models can be used to identify and

schedule webpages using the ideal core and frequency configuration to minimize energy consumption while still meet a stringent cut-off constraint.

Radio Resource Optimizations: Qian et al [26] initially began work into accurately inferring the RRC state machine of any 3G UMTS network. Based on this they then pinpointed the inefficiencies caused by the interplay between smartphone applications and the state machines behaviour. They then [23] designed and implemented ARO (Application Resource Optimizer), the first tool to accurately expose the interactions between the various levels of the software stack. Primarily, the tool allows a user to discover inefficient RRC resource usage in smartphone applications. Finally [25], using the ARO tool, they observed that periodic transfers were at the core of radio resource and energy efficiencies in smartphone applications. Furthermore, they investigate the potential of traffic shaping and resource control algorithms to optimize periodic transfers.

Chapter 7 – Conclusion

We have presented a DVFS energy-aware approach for web browsers on 3G UMTS and Wi- Fi networks. The proposed approach consists of two components; the partitioned browser architecture and a cloud-assisted DVFS algorithm. The partitioned browser architecture reorganizes the workflow of a web page load into two non-repeating phases, loading and rendering. The loading phase parses, downloads all resources and sub resources from the network/cache and constructs the DOM tree, whereas, the rendering phase paints the web page onto the display. The phases are non-repeating as only once all resources have been downloaded and the DOM tree has been constructed can the second phase of rendering begin. The partitioned browser architecture allows the 3G radio interface to spend less time in the high power DCH RRC state and move into a lower power IDLE RRC state quicker than the original browser architecture. The proposed approach not only saves power in the 3G network adapter of the smartphone device but also benefits the backbone networks' resource utilization.

The cloud-assisted DVFS algorithm capitalizes on the use of the partitioned browser architecture by operating at two frequencies; a loading and rendering frequency. As the loading phase of the partitioned browser is an IO bound phase the loading frequency operates at a low frequency of 500Mhz. The

rendering phase of the partitioned browser operates at an optimal frequency. The optimal frequency is a CPU frequency that can perform the rendering of the web page in at least the same time as the default rendering frequency; determined by the ONDEMAND governor. The calculation of the optimal frequency takes place in the cloud concurrently to the loading phase. To calculate the optimal frequency the cloud uses web page characterization and regression models to predict the rendering time and energy consumption of the web page at each available frequency, before choosing an optimal which at least matches the default frequencies' rendering time with less energy consumption. The cloud-assisted DVFS algorithm allows us to save CPU power in both loading and rendering phases and still completes the web page load in at least the same time as the stock original browser. Experimental results show that our approach can reduce the power consumption of a web page load on a smartphone device by 12% in a Wi-Fi environment and 14.5% in a UMTS 3G network environment with no added latency to web page load times. Moreover, in a 3G network environment the proposed approach can also increase 3G radio resource utilization by 9%.

Chapter 8 - References

[1] Alexa top 500 sites. *http://www.alexa.com/topsites*.

[2] Always connected: How smartphones and social keep us engaged: Idc research report. *https://fb-public.box.com/s/3iq5x6uwnqtq7ki4q8wk*.

[3] Android. *http://developer.android.com/index.html*.

[4] At&t application resource optimizer. *http://developer.att.com/developer/forward.jsp?passedItemId=9700312*.

[5] Cyanogenmod 9.1.0. *http://wiki.cyanogenmod.org/w/I9100_Info*.

[6] Danes least satisfied with smartphone battery. *http://www.telecompaper.com/news/danes-least-satisfied-with-smartphone-battery-survey–942766*.

[7] Monsoon power monitor. *http://www.msoon.com/*.

[8] Netmarketshare. *http://netmarketshare.com/*.

[9] Network link conditioner. *https://developer.apple.com/*.

[10] Nokogiri html parser. *http://nokogiri.org/*.

[11] Optimizations for the mobile web: Defer parsing of javascript. *https://developers.google.com/speed/docs/best-practices/mobile*.

[12] Ruby css_parser. *https://github.com/alexdunae/css_parser*.

[13] suppressincrmentalrendering webkit source change. *http://git.chromium.org/gitweb/?p=external/WebKit_trimmed.git;a=commitdiff;h=98dc1617b39814720f9866136bb91435a20b7863*.

[14] Webkit. *http://www.webkit.org/*.

[15] Aaron Carroll and Gernot Heiser. An analysis of power consumption in

a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[16] Talis Garsiel and Paul Irish. How browsers work: behind the scenes of modern web browsers. *http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/*, August 2012.

[17] H. Holma and A. Toskala. *HSDPA/HSUPA for UMTS: high speed radio access for mobile communications*. John Wiley, 2006.

[18] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodk. Parallelizing the web browser. Ín *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association.

[19] Wen-Yew Liang and Po-Ting Lai. Design and implementation of a critical speed-based dvfs mechanism for the android operating system. In *Embedded and Multimedia Computing (EMC), 2010 5th International Conference on*, pages 1–6, 2010.

[20] Farhad Manjoo. Have smartphone batteries reached their peak? *http://www.independent.co.uk/life-style/gadgets-and-tech/features/power-out-have-smartphone-batteries-reached-their-peak-8586830.html*.

[21] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[22] P.H.J. Perala, A. Barbuzzi, G. Boggia, and K. Pentikousis. Theory and practice of rrc state transitions in umts networks. In *GLOBECOM Workshops, 2009 IEEE*, pages 1 –6, 30 2009-dec. 4 2009.

[23] Feng Qian, Z. Morley Mao, Zhaoguang Wang, Subhabrata Sen, Alexandre Gerber, and Oliver Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *In Mobisys*, 2011.

[24] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 127–140, New York, NY, USA, 2012. ACM.

[25] Feng Qian, Zhaoguang Wang, Yudong Gao, Junxian Huang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 51–60, New York, NY, USA, 2012. ACM.

[26] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 137–150, New York, NY, USA, 2010. ACM.

[27] Jordi Perez Romero, Oriol Sallent, Ramon Agusti, and Miguel Angel Diaz-Guerra. *Radio Resource Management Strategies in UMTS*. John Wiley & Sons, 2005.

[28] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 41–50, New York, NY, USA, 2012. ACM.

[29] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 91–96, New York, NY, USA, 2011. ACM.

[30] Bo Zhao, Qiang Zheng, and Guohong Cao. Energy-aware web browsing in 3g based smartphones. In *Internal Technical Report*. The Pennsylvania State University, 2011.

[31] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *The 19th IEEE International Symposium on High Performance Computer Architecture*, HPCA-2013, 2013.

초록

최근 들어 스마트폰을 통한 웹 브라우저 사용이 보편화되면서 스마트폰의 웹 브라우저에 여러 가지 기능이 추가되었으며, 스마트폰 웹 브라우저 에너지 소비가 전체 스마트폰 전력 소비에서 큰 부분을 차지하게 되었다. 이에 따라서, 본 논문에서는 웹 브라우저 사용에 따른 스마트폰의 전력 소비를 감소시키기 위해서 다음과 같은 두 가지 기술을 제안한다. 첫 번째로 우리는 다양한 웹 페이지 요소를 위해 무작위로 동작되는 웹 브라우저의 작업 흐름을 묶어 두 분류(입출력 바운드 작업과 중앙처리장치 바운드 작업)로 나누어서 네트워크 어댑터를 유휴 상태로 빠르게 변경 시킴으로써 네트워크 어댑터의 에너지 소비를 감소시켰다. 두 번째로 우리는 동적 전압 주파수 스케일링 기반 알고리즘을 적용함으로써 중앙처리장치의 에너지 소비를 감소시켰다. 동적 전압 주파수 스케일링 기반 알고리즘은 중앙처리장치가 웹 페이지 로드(입출력 바운드 작업) 시에는 낮은 주파수로 동작하게 하고, 렌더링(중앙처리장치 바운드 작업) 시에서는 최적의 주파수로 동작하게 설계되었다. 최적의 주파수는 전통적인 중앙처리장치 전압 주파수 관리 방법과 동일한 속도로 렌더링 작업을 수행하면서, 더불어 에너지 소비를 감소시킬 수 있는 주파수로 선정된다. 우리는 주파수 선정은 스마트폰이 웹 페이지를 로딩할 시에 클라우드를 통해서 계산되고, 렌더링 시에 스마트폰의 중앙처리장치에 적용될 수 있도록 설계하였다. 클라우드는 웹 페이지의 구성 요소 분석과 회귀분석 모델을 통하여 렌더링을 위해 스마트폰을 위한 최적의 주파수를 예측한다. 본 논문에서는 제시된 에너지 소비 감소법을 실험하

기 위해서 안드로이드 OS를 탑재한 삼성 갤럭시 S2를 사용하였다. 실험 결과, 전통적인 웹 브라우저 보다 더 긴 웹 페이지 로드 대기 시간을 갖지 않았으며, 무선랜 사용 시에는 웹 페이지 로드의 전력 소비가 12%가 감소하였고, 3G 네트워크 사용 시에는 웹 페이지 로드의 전력 소비가 14.5%가 감소하였다.