Who Killed My Battery: Analyzing Mobile Browser Energy Consumption

Narendran Thiagarajan[†] naren@cs.stanford.edu
Gaurav Aggarwal[†] agaurav@cs.stanford.edu
Angela Nicoara[*] angela.nicoara@telekom.com
Dan Boneh[†] dabo@cs.stanford.edu
Jatinder Pal Singh[‡] jatinder@stanford.edu

[†]Department of Computer Science, Stanford University, CA
[*]Deutsche Telekom R&D Laboratories USA, Los Altos, CA
[‡]Department of Electrical Engineering, Stanford University, CA

ABSTRACT

Despite the growing popularity of mobile web browsing, the energy consumed by a phone browser while surfing the web is poorly understood. We present an infrastructure for measuring the precise energy used by a mobile browser to render web pages. We then measure the energy needed to render financial, e-commerce, email, blogging, news and social networking sites. Our tools are sufficiently precise to measure the energy needed to render individual web elements, such as cascade style sheets (CSS), Javascript, images, and plug-in objects. Our results show that for popular sites, downloading and parsing cascade style sheets and Javascript consumes a significant fraction of the total energy needed to render the page. Using the data we collected we make concrete recommendations on how to design web pages so as to minimize the energy needed to render the page. As an example, by modifying scripts on the Wikipedia mobile site we reduced by 30% the energy needed to download and render Wikipedia pages with no change to the user experience. We conclude by estimating the point at which offloading browser computations to a remote proxy can save energy on the phone.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software Engineering**]: Software Architectures; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*

General Terms

Design, Measurement, Performance

Keywords

Mobile browser, Energy consumption, Offloading computations, Android

1. INTRODUCTION

Recent statistics from NetMarketShare show that about 3% of *all* worldwide web browsing is done on mobile browsers [5]. Many popular sites responded by providing a mobile version of their site optimized for a small screen. However, we show that many mobile sites are poorly optimized for energy use and rendering them in the browser takes more power than necessary. Partly this is due to a weak understanding of the browser's energy use.

In this paper we set out to analyze the energy consumption of the Android browser at popular web sites such as Facebook, Amazon, and many others. Our experimental setup includes a multimeter hooked up to the phone battery that measures the phone's energy consumption as the phone loads and renders web pages. We patched the default Android browser to help us measure the precise energy used from the moment the browser begins navigating to the desired web site until the page is fully rendered. The patch also lets us measure the exact energy needed to render a page excluding the energy consumed by the radio. Our setup is described in detail in Section 2. In that section we also describe the energy model for the phone's radio which is similar to models presented in [18, 10].

Using our experimental setup we measured the energy needed to render popular web sites as well as the energy needed to render individual web elements such as images, Javascript, and Cascade Style Sheets (CSS). We find that complex Javascript and CSS can be as expensive to render as images. Moreover, dynamic Javascript requests (in the form of `XMLHttpRequest`) can greatly increase the cost of rendering the page since it prevents the page contents from being cached. Finally, we show that on the Android browser, rendering JPEG images is considerably cheaper than other formats such as GIF and PNG for comparable size images. For example, when we translate all images on the Facebook web site to JPEG we obtain considerable energy savings.

Using our energy measurements we suggest guidelines for building energy-efficient web pages, namely pages that reduce energy use on the client. For example, by applying our guidelines to the Wikipedia mobile site we reduced its energy consumption from 35 Joules to 25 Joules, a saving of 29%. Our modification simply changes how Javascript works on the page, without affecting the user experience. The measurements in this paper quantify how much energy can be saved by following these guidelines.

Beyond optimization, our experiments let us estimate the effectiveness of offloading browser computations to a remote server. Section 5 gives quantitative numbers for a modern smartphone, the Android ADP2 phone [2]. We discuss related and future work in Sections 6 and 7. To promote further research on building "green" energy efficient web sites we plan to release our experimental setup and measurement code for others to use.

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

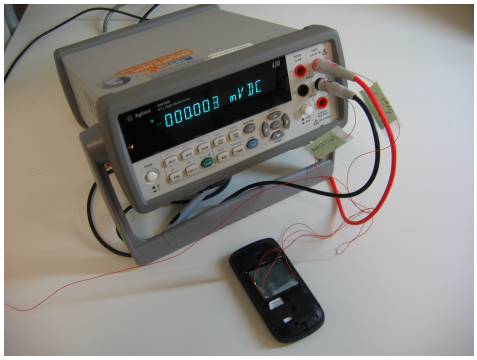


Figure 1: The hardware power multimeter and an open battery used for measuring energy consumption

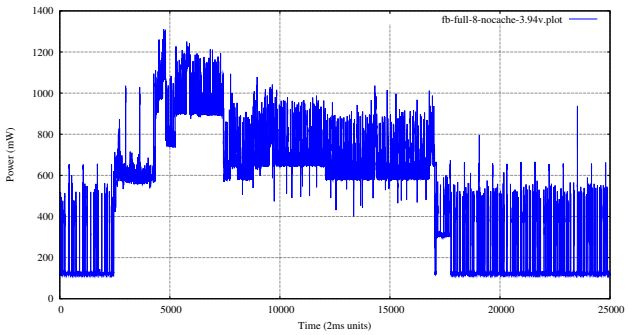


Figure 2: Sample multimeter output graph for a Facebook page

2. METHODOLOGY

We begin by describing our hardware and software setup.

2.1 Hardware Setup

Our experiments were performed on an Android Developer Phone 2 (ADP2) [2]. The mobile device is 3G-enabled T-Mobile phone that uses 3G and 2.5G and is equipped with an ARM processor, 192MB/ 288MB RAM, a 2GB MicroSD card, and an 802.11b/g WiFi interface. We measure its battery capacity in Section 2.4.

Today's mobile devices support a high level API for finding out the battery level, but provide no support for obtaining precise fine-grained energy use. To obtain precise measurements we use the Agilent 34410A [1] high-precision digital power multimeter shown in Figure 1. The multimeter provides fine grained measurements every 1 milliseconds, namely a sampling rate of 1kHz. A sample power graph is presented in Figure 2, where the high power interval captures browser activity.

The Android mobile device will not boot without the battery in the phone. Therefore, we left the battery inside the phone and measured continuous power transferred from the battery to the phone. The charger was disconnected in order to eliminate interference from the battery charging circuitry. To measure the energy consumption, we opened the battery case and placed a 0.1 ohm resistor in series with the ground. We measured the input voltage to the phone and the voltage drop on the resistor from which we calculate the phone's instantaneous power consumption.

2.2 Software Setup

In addition to the hardware setup we also had to modify the default Android browser. Our modified browser enables us to fully load a URL P in one of two modes:

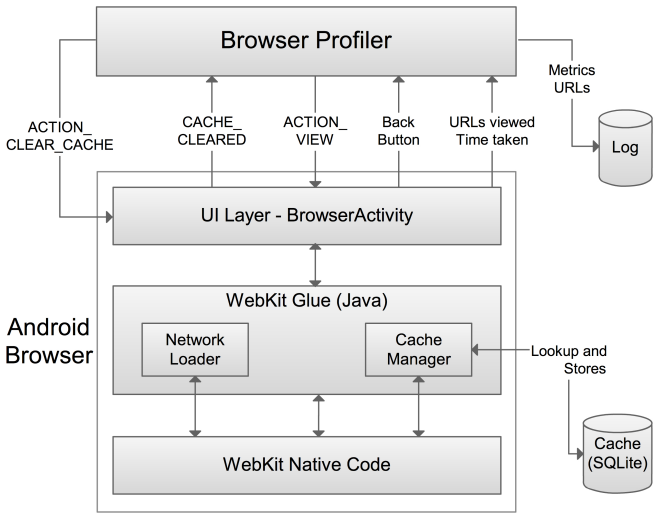


Figure 3: System architecture

- **No Cache**. Browser cache is emptied before starting to load the URL so that all elements of the web page are first downloaded from the network and then rendered on the phone. This mode lets us measure the total energy used for navigating to the page, including 3G transmission, parsing HTML, and rendering.
- **With Cache**. All elements of the web page are already present in the browser cache so that there is no need to use the radio to download any content. This mode lets us measure the energy needed to parse and render HTML from cache. No 3G traffic is allowed in this mode.

Our software setup consists of two components: (1) a *Browser Profiler*, an Android application we wrote, and (2) the built-in *Android Browser* with some modifications described below. We will refer to these as *Profiler* and *Browser* respectively. Figure 3 illustrates the information flow between these components.

Measurement Workflow. The Profiler provides a simple user interface that takes URL P and number of iterations n as input.

When the user taps a button to start profiling, Profiler tells the browser to load the web page P in `NoCache` mode. First, Profiler instructs the browser to clear its cache by sending `ACTION_CLEAR_CACHE` intent [4]. Browser responds by completely clearing its cache and sends back `CACHE_CLEARED` intent to Profiler. Both these intents are custom intents defined by us and discussed in detail later in this section. Now, Profiler asks the browser to load web page P by sending the built-in `ACTION_VIEW` intent. Once page load finishes, user presses the `BACK` button on the Android device to transfer control back to Profiler. This process is repeated n times and represents n page loads of P in `NoCache` mode.

At the end of `NoCache` mode, all components of page P will be present in the browser cache. Now, Profiler asks the browser to load P again n times using same combination of `ACTION_VIEW` intent and `BACK` button as before. However, we do not clear the cache after every load this time. So, this represents n page loads of P in `WithCache` mode.

At the end of every page load, Profiler also logs the following information to a file:

1. WiFi and 3G signal strength obtained using Android API.

2. Tx/Rx bytes: Number of bytes sent and received by the browser obtained using `getUidTxBytes(int uid)` and `getUidRxBytes(int uid)` functions in `android.os.NetStat` class.

3. Page load time and list of URLs corresponding to all components of the page that were downloaded. This information is sent by the browser after page load completes.

Changes to the Android Browser. At a high level, the Android Browser consists of three layers:

1. **UI Layer** - This contains all Android Activities [3] that comprise browser chrome, user interface elements like buttons, menus and corresponding event handlers.
2. **WebKit Glue** - This contains Java classes that maintain current state of the browser like open tabs and acts as an intermediary between the UI Layer and native WebKit. It includes `CacheManager` class which provides an interface to store and lookup pages from a SQLite based cache. Also, `NetworkLoader` class is used to download content from the network.
3. **Native WebKit Layer** - This consists of the native WebKit [8] library which parses and renders downloaded web pages on the phone screen. It relies on `NetworkLoader` and `CacheManager` classes to download different components of a web page. This layer also contains the Javascript engine.

To measure the precise energy used by the browser we had to make a few modifications to the default Android browser.

1. **Cache management**. As described earlier, we load a page in `WithCache` and `NoCache` mode to isolate the energy used for rendering a web page from that used for transmission. To implement these modes we had to make the following changes to cache management in the Android browser:

   - *Cache everything* - `WithCache` mode can be used to measure rendering energy only if all components of the web page are cached and hence there is no need to download any new content. `CacheManager` class contains the browser cache management policy. We modified this class to cache redirects containing a cookie header and HTTP responses with zero content length which are otherwise not cached. Also, we ignore Cache-Control headers, `Pragma: no-cache` and `Expires` header field in any HTTP response.
   - *Clear cache programmatically* - Browser contains a Preference option to clear the cache that can only be set manually. Since we would be using Profiler to make measurements, we introduced a new Android intent called `ACTION_CLEAR_CACHE`. As described earlier, Profiler issues this intent to the browser which acts upon it by clearing its cache. We also added another intent `CACHE_CLEARED` to serve as a callback from the browser to Profiler to inform that cache has been cleared so that it can continue with the next measurement.
   - *Handle "changing" URLs* - During our experiments we noticed that despite our caching mode, the browser was still downloading content from the network. Part of the reason was due to GET requests with varying parameters. For example, while loading `www.google.com`, the browser downloads `http://www.google.com/m/gn/loc?output=json&text=87135` and caches the result. However, when loading the same page from cache, the browser tries to download a slightly different URL - `http://www.google.com/m/gn/loc?output=json&text=94219`. Since the value of the `text` parameter is different from the cached copy, cache lookup fails and a network request is issued. As a workaround, we identified all such "changing" URLs and modified browser code to ignore the GET parameters in cache lookup. This suppressed network traffic for these pages. [Note that we could not simply turn off the 3G radio, since then the page would not render].

2. **Intercept Page Load**. When recording samples using multimeter as explained in Section 2.1, there should be minimal interaction with the phone to ensure accurate measurements. To achieve this, we modified the browser to ask user's permission to start loading a web page by displaying a dialog and suspend all browser activity until it obtains the permission. Now to take measurements we can follow this simple process: Enter URL to measure and hit load on the browser. Browser will display the dialog and wait for user to press "ok". Then, setup the multimeter and trigger recording of samples. Lastly, hit "ok" on the dialog to start loading the web page.

3. **Track metrics**. Browser keeps a list of all component URLs that are downloaded over the network as part of rendering the page. It also tracks the time taken to load the entire web page. Once the page load completes, it sends this list of downloaded URLs and page load time to Profiler which logs them to a file. When page is being loaded from cache, this list of URLs should be empty.

Impact on measurements. Our modifications to the browser do not change the browser's energy profile. Clearing the cache programmatically does not modify the code path for loading a web page. Changes to cache everything mainly involve commenting out code that checks for Cache-control headers, `Expiry` header field etc. Our other modifications such as, tracking metrics and string comparisons during cache lookup have negligible impact on browser energy use.

2.3 Automation

The energy measurement system described in Section 2.2 requires significant user assistance in the measurement process. To perform a large number of measurements we automated the process by using the SCPI programming interface on the multimeter.

System components. Our energy measurement system consists of the following components:

- *Server*,
- *Android Phone (client)*, and
- *Multimeter*.

Figure 4 shows how these components interact. The server controls the phone (client) and the multimeter, telling it when to start, stop and save the measurements. When the experiment starts the server communicates with our Browser Profiler application on the phone. The server instructs this application to request the running phone browser to repeatedly load a specific URL, either with or without caching.

During this process the server also starts the multimeter measurement. The server communicates with the Agilent multimeter using SCPI commands. Due to the limitations set by SCPI commands, the highest sampling rate achievable for the measurements is 2.5 kHz (i.e. 2500 measurements per second). At the end of

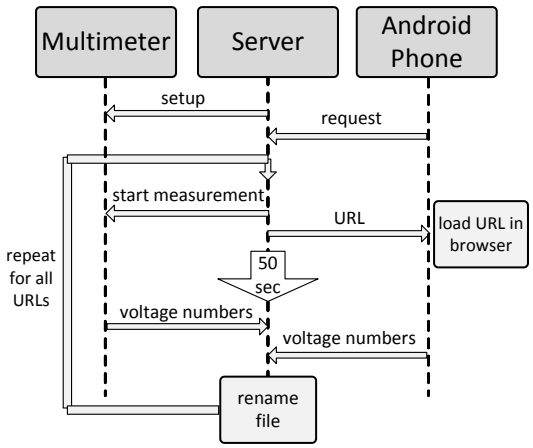


Figure 4: Automated energy measurement system

the experiment, all measurements recorded on the multimeter are transfered to the server for processing.

Software changes. The browser is modified to contain a single tab so that all loads take place on the same tab. After loading each URL the browser is navigated to an empty page so that all measurements start with the browser in the same state. The server-client communication takes place over 3G and not USB because connecting the phone to a computer via USB starts charging the phone, thereby rendering the measurement unusable.

2.4 Battery Capacity Measurement

All our energy measurements are stated in Joules or millijoules. To make sense of these numbers we often state energy used as a fraction of the total battery capacity.

To determine the battery's energy capacity in Joules we performed the following simple experiment. We ran the multimeter for 250 seconds sampling the power consumption every 5 milliseconds. During these 250 seconds we stressed the phone by constantly browsing random web pages. At the end of the 250 seconds we observed a total energy use of 229.88 Joules that resulted in a 7% drop in battery charge. From this experiment we learned the following important fact:

1% of a fully charged battery is
approximately 32.84 Joules.

2.5 3G Radio Energy

To better understand the energy consumed by the 3G radio we measured the energy needed to setup a 3G connection with the base station and the energy needed for varying payload sizes.

Figure 5 shows the average energy needed for downloading or uploading 4, 8, 16, 32, 64, 128, and 256 kB over 3G. All measurement results are averaged over five runs and the standard deviation is less than 5%. In all measurements the display brightness is set to the minimum level. We measured the average idle power on the device and found it to be 170 millijoules per second. We subtract this number from all our measurements.

Energy model. Figure 5 shows two important facts about 3G behavior. First, for both download and upload there is a high setup cost of roughly 12 Joules before the first byte can be sent. Second, once the connection is established, the energy needed to download data is mostly flat, which means that roughly the same energy is used no matter how much data is downloaded (up to 256KB). The situation is a little different for uploading data where the energy

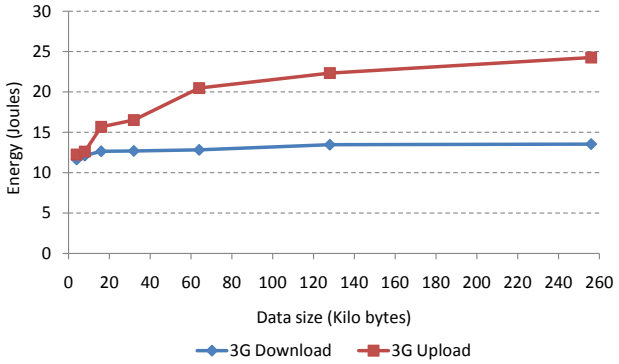


Figure 5: Energy consumption over 3G (download and upload)

does increase with the amount of data being uploaded. These observations are consistent with earlier work [18, 10].

The experiment used to generate the data in Figure 5 separates upload activity from download activity. That is, the upload numbers in the figure are obtained by sending data from the phone and not using the radio for any other task. The download numbers are obtained similarly. A web browser, however, simultaneously sends HTTP requests and receives HTTP responses using multiple threads as it requests and receives the various components that make up a web page.

Naively one might conjecture that the energy needed to send 16kB and then receive 16kB is the sum of the energies for uploading and downloading 16kB shown in Figure 5. But this turns out to be false. Our experiments show that a mild interleaving of uploads and downloads is essentially dominated by the cost of the upload plus a relatively small quantity. More precisely, suppose we upload and then download a 1kB chunk and repeat this process *eight* times (that is, we upload a total of 8kB and download a total of 8kB at 1kB per iteration). Then the total energy used is only 5% more than the energy needed to directly upload 8kB of data. Hence, the upload and download energies in Figure 5 should not be summed to estimate the radio energy used by a web page. Instead, for mild interleaving the cost is dominated by the upload energy. Note that the 3G setup cost is only incurred once.

When we repeated the experiment with a larger number of repetitions (256) the energy used by the radio grew to more than the sum of the corresponding upload/download energies. This suggests, and our experiments confirm, that Figure 5 should not be used when there are many round trips.

In summary, Figure 5 can be used to model applications that mostly use one-way communication such as streaming video. For Web traffic, that interleaves uploads and downloads, it can be quite difficult to define an energy model that accurately predicts the 3G energy needed to fetch a web page since energy use depends on the precise shape of the traffic. We do not use a model to estimate the radio energy needed to fetch a web page. All our data is derived from experiments.

3. MEASUREMENTS

Using the infrastructure described in the previous section we obtain insights on the energy consumption of mobile web sites and web elements. We describe our experiments and findings here.

3.1 Energy Consumption of Top Web Sites

Our first experiment measures the energy consumption used by web pages at several top sites. We chose sites representing

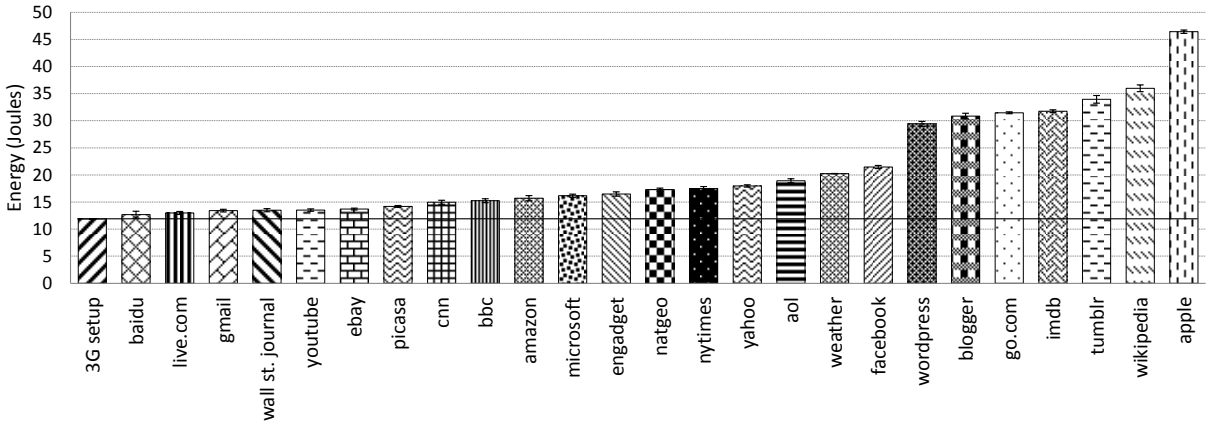


Figure 6: Energy consumption of top websites

Web site	Comment	% Battery life	Traffic (bytes)	
			Upload	Download
m.gmail.com	inbox	0.41	9050	12048
m.picasa.com	user albums	0.43	8223	15475
m.aol.com	portal home	0.59	11927	37085
m.amazon.com	product page	0.48	9523	26838
mobile.nytimes.com	US home page	0.53	15386	66336
touch.facebook.com	facebook wall	0.65	30214	81040
mw.weather.com	Stanford weather	0.62	38253	134531
apple.com	home page	1.41	86888	716835
m.imdb.com	movie page	0.97	30764	127924
m.microsoft.com	home page	0.49	15240	47936
m.natgeo.com	home page	0.53	13877	76742
m.wikipedia.org	article page	1.09	43699	308832
bbc.com	mobile home page	0.46	20505	67004
m.ebay.com	product page	0.42	8041	17941
m.yahoo.com	portal home	0.55	14397	45564
m.youtube.com	home page	0.55	5704	20329
baidu.com	search page	0.39	2108	3951
blogger.com	home page	0.94	45382	427788
m.cnn.com	headlines page	0.46	9311	33844
m.engadget.com	portal page	0.50	23334	80432
m.go.com	start page	0.96	27965	154278
m.live.com	personal page	0.40	7319	12576
wordpress.com	home page	0.90	23318	205140
tumblr.com	home page	1.03	40543	889242
m.wsj.com	news page	0.41	4058	13653

Table 1: Web sites used in measuring energy consumption

e-commerce, social networking, email, blogging, portals, news, videos, product and financial sectors. The complete list of sites is shown in Table 1 along with the amount of traffic in bytes needed to request and download the page. A summary of the energy consumed by these sites is shown in Figure 6. Table 1 also shows the energy needed to download and render the page as a fraction of a fully charged battery (computed using the battery measurements from Section 2.4).

Experiment details. To measure the total energy used to download and render the page we first measured the phone's average energy consumption when the browser is idle, which is 170 mJ/sec. Then the web pages to be measured are downloaded and saved in a remote server running Apache web server. We then used our Browser Profiler application to measure the energy consumption from the moment the browser begins processing the navigation request until the page is fully rendered. Each measurement was repeated up to ten times. The difference between the idle energy measurement and the energy when processing the request is the (average) total energy used to download and render the page. This includes the energy needed for 3G communication and for parsing and rendering the page, but does not include the phone's idle energy consumption. The resulting numbers are shown in Figure 6. Note that the error bars are so small that they are barely visible.

The left most column in Figure 6 shows the energy needed to set up a 3G connection and download a few bytes without any additional processing. Since all navigation requests must setup a 3G connection we treat this measurement as a baseline where the interesting differences between web sites are above this line.

Figure 6 is generated from the mobile versions of the web site shown. The exceptions are `apple.com`, `tumblr.com`, `blogger.com` and `wordpress.com` that do not have a mobile web site. As a result, the amount of data needed to retrieve these sites is significantly higher than for other web sites. For example, Apple's page contains many images, including a 26kB welcome image that is mostly wasted on the phone since the phone scales it down to fit its small screen.

Rendering energy. Next we measure the energy needed to parse and render the page without the energy consumed by the radio. That is, we determine how the complexity of the web page affects the energy needed to render it.

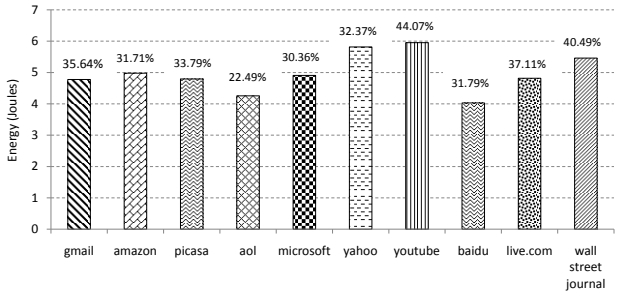
To measure the rendering energy we forced the browser to locally cache all web content and then measured how much energy was used to render the content from local cache. We made sure that:

1. There was no network traffic while rendering from local cache, and
2. The cached data was identical to data fetched from the web site, that is, the browser did not pre-process the data before caching it.

Consequently, this experiment measures the energy used to parse and render the page when all contents are already in memory. The resulting numbers are shown in Figure 7. The percentages above the bars show the energy to render the page as a fraction of the total energy to download and render the page.

We only include measurements for 10 of the 25 sites in Figure 6. For the remaining 15 sites caching web content did not prevent network traffic. Javascript and CSS at these sites generated dynamic web requests that could not be cached ahead of time. On the Apple home page, for example, the Javascript used for tracking user location generates an update forcing the phone to setup a 3G connection. Thus, despite caching, energy consumption for these 10 sites was almost as high as when no caching took place. There is an important lesson here for mobile web site design — dynamic Javascript can greatly increase the power usage of a page. We discuss this issue further in Section 7.

Analysis. Figure 7 shows that mobile sites like baidu, that are mostly text and very little Javascript and no large images, consume



(Percentages show energy relative to Figure 6)

Figure 7: Rendering energy of top websites from cache

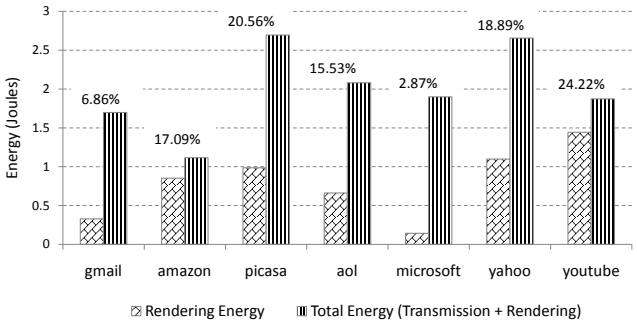


Figure 8: Energy consumed by images. The percentage numbers refer to the total energy needed to render images as a fraction of the energy for rendering the entire page.

little energy to render. The Amazon site that contains a product image take more energy. Sites like youtube and yahoo that contain images, Javascript, and CSS take considerably more energy to render. We study the precise reason for these differences in the next section.

3.2 Energy Consumption of Web Components

Next, we look at the energy consumption of individual web elements such as images, Javascript, cascade style sheets (CSS), etc. The question is how much energy is used by different web elements.

To measure the energy used by a particular element we created a copy of the web page on our servers and then compared the energy consumption used for loading and rendering the entire page to the energy consumption needed for loading and rendering the page with the particular component removed by commenting it out. The difference between the two numbers gives an estimate for the energy needed to present the component. These experiments were run on web sites that contain specific components (Figure 7). For example, our results for Javascript are illustrated in Figure 9.

As in the previous section we first measured the total energy used for loading and rendering each component, which includes both rendering and transmission energy. We then measured parsing and rendering energy alone by forcing the browser to cache all content locally on the phone.

Images. Figure 8 shows energy measurements for images on web pages. The bars on the right correspond to the total energy spent on images. The bars on the left show the rendering energy for cached images. The percentage above the left bar shows the energy spent on rendering images as a fraction of the energy for rendering the entire page from cache.

As expected, rendering images takes a significant fraction of the

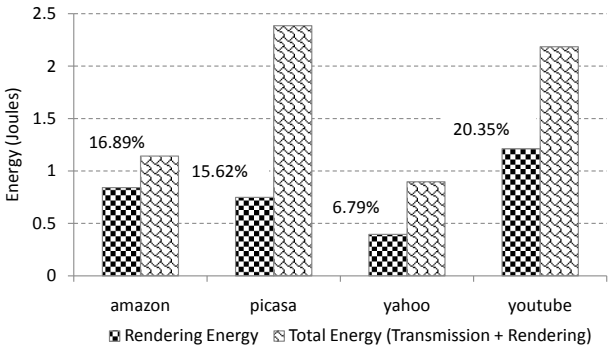


Figure 9: Energy consumption of Javascript

total rendering energy. Some sites like Youtube spend around quarter of their rendering energy on images.

The amount of energy used to render images is proportional to the number and size of images on the page. The Youtube page has 5 large images representing a screenshot from each video which is why 24.22% of the total energy to render the page is spent on images. Gmail, in contrast, contains only small GIFs (13 pixels wide) and images take a smaller fraction of the total energy (6.86%). We found that one small GIF on the Gmail page is repeated 16 times. This GIF indicates whether an email was sent to a single recipient or a group.

Picasa spends a large fraction of its rendering energy (20.56%) on images because the user album page contains 8 large album cover images.

Javascript. Figure 9 shows similar measurements for Javascript on web pages. Of the cachable websites that we considered only Amazon, Picasa, Youtube and Yahoo have Javascript.

Amazon consumes 16.89% of its rendering energy for handling Javascript. The reason for the large rendering cost is a large and complex Javascript file. Many of the Javascript functions in the file are not used by the page but loading and processing the entire Javascript file consumes a lot of energy.

Yahoo's Javascript code is embedded in the HTML page. The amount of Javascript code is very small but the code gets executed every time the page loads. As a result Javascript processing takes only 6.79% of the total rendering energy. The Javascript code here is minimal and fully used in the page. Youtube's Javascript is embedded but is so heavy that it takes 20.35% of the total energy.

Cascade Style Sheets (CSS). Finally, Figure 10 shows energy measurements for CSS. The rendering cost of CSS depends on the number of items styled using CSS. Amazon has a high CSS rendering cost (17.57%) since about 104 items in the page use styling defined in the CSS file. Amazon also has complex styling in the page like a color fade out effect, horizontal bars to show product ratings, and huge wide buttons. Gmail has simple styling defined in an internal style sheet. The CSS is very small and it consumes only 3% of the total rendering energy.

AOL and Picasa both contain large images but the CSS energy consumption for AOL is much lower than Picasa's. The reason is that AOL uses HTML tables to position its images while Picasa uses CSS to position images. This nicely illustrates how positioning using CSS is less energy efficient than positioning using simple HTML tags.

Microsoft and Yahoo pages use large CSS files that causes a very high transmission energy cost to download the file.

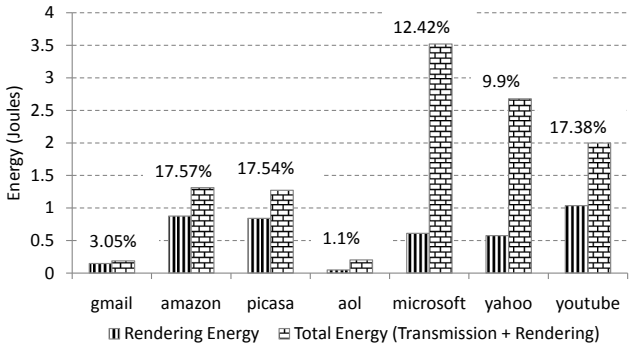


Figure 10: Energy consumption of cascade style sheets

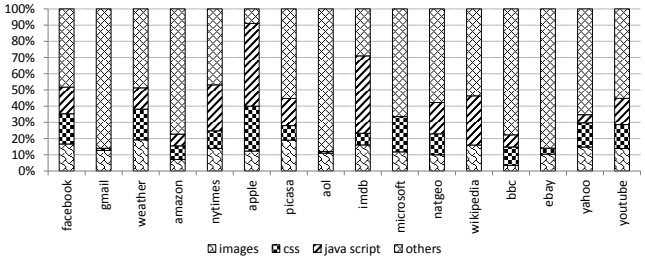


Figure 11: Total energy consumption of web components (Transmission + Rendering)

The relative energy used on web components. For completeness, Figure 11 shows the relative costs of individual components. The general trend across web sites is that CSS and Javascript are the most energy consuming components in the transmission and rendering of a site. The value of 'others' in the graph mainly includes the 3G connection setup cost and text rendering.

Sites that have high 'others' (such as AOL, Ebay and Gmail) can become more efficient as wireless technology improves and connection setup cost decreases. Sites with low 'others' (such as Apple and IMDB) spend much of their energy on web elements and will not gain much from improvements in wireless technology. Thus, Figure 11 is another method for comparing web site efficiency.

4. OPTIMIZING MOBILE WEB PAGES

Power hungry web components include images, Javascript, and CSS. In this section we show how to optimize web pages so as to reduce the power consumption of these elements.

4.1 Reducing Javascript Power Consumption

Javascript is one of the most energy consuming components in a web page. Figure 9 shows a high download and rendering energy required by most of the websites for Javascript. This is mainly because these webpages load large Javascript files for rendering the web page even though not all of the script is used by the page. For example, the download and rendering of Javascript in the Wikipedia page takes about 10 Joules. This is about 30% of the total energy to download and render the page.

The Wikipedia webpage has two Javascript files linked to the page - *application.js* and *jquery.js*. The *application.js* is the Javascript specific to the Wikipedia site and the *jquery.js* is the generic jquery Javascript library. In the Wikipedia page each section of the page like Introduction, Table of Contents, etc. can be collapsed and expanded by the click of a button above each section. The Javascript in *jquery.js* is used primarily for a single purpose - to dynamically identify the correct section based on the id of the button clicked. But loading this Javascript to the memory alone takes 4 Joules.

In order to prove that this energy is avoidable, we redesigned the page with a different Javascript. This time each text section and the button are given the same id and the Javascript function uses document.getElementById() to get the right section and element.value=show/hide is used. The *application.js* is now replaced by this simple Javascript. We found that in cache mode, the modified Wikipedia page renders with 9.5 Joules. Just adding the *application.js* and *jquery.js* files as link to the page increases the energy consumption to 15 Joules.

This experiment shows that shrinking Javascript on a mobile page to contain only functions used by the page greatly reduces energy use. Using generic Javascript libraries simplifies web development, but increases the energy used by the resulting pages.

4.2 Reducing CSS Power Consumption

Similar to the previous experiment we found that large CSS files with unused CSS rules consume more than minimum required energy. For example, Apple consumes a large amount of energy to download and render CSS (Figure 10). The total energy to download and render CSS of this page is around 12 Joules. This is because the Apple home page requires 5 different CSS files containing different rules used in the page.

We modified the Apple site by replacing multiple CSS files with just one CSS file containing just the rules required by the page. This resulted in an energy drop of 5 Joules. This is about 40% of the total CSS energy consumed by Apple. This energy can be saved by using a CSS file with only the required CSS rules.

This shows that like Javascript, CSS file should be page specific and contain only the rules required by the elements in the page.

4.3 Image Formats: Comparison and Optimization

The web sites we analyzed use a variety of image formats, with JPEG, GIF, and PNG being the most common. Since the energy needed to render an image depends on the encoding format we set out to compare the energy signature for different formats. We focus on these three predominant formats.

Recall that the GIF format supports 8-bits per pixel and uses the Lempel-Ziv-Welch (LZW) lossless data compression method. PNG is similarly a bitmapped image format that was created to improve upon and replace GIF. PNG also uses lossless data compression. JPEG is another popular image format using lossy data compression.

On the mobile web sites we examined GIFs were mostly used for very small images such as small arrows and icons, PNGs were used for larger images such as banners and logos, and JPEGs were used for large images.

Image formats for different dimensions. Figure 12 shows the energy consumption needed to download and render images of different sizes in the three formats on the Android phone.

This experiment used a JPEG image of dimensions 1600x1200 and size 741kB. Smaller images of different height and width are cropped from this image as shown on the x-axis. We then saved the cropped images as JPEG, GIF and PNGs. Each image was then embedded in a web page that contained the image and nothing else. Energy needed to download and render each image is measured for all sizes shown on the x-axis with the energy along the y-axis. Since GIF and PNG are only used for small images, we only experimented with these formats for small images.

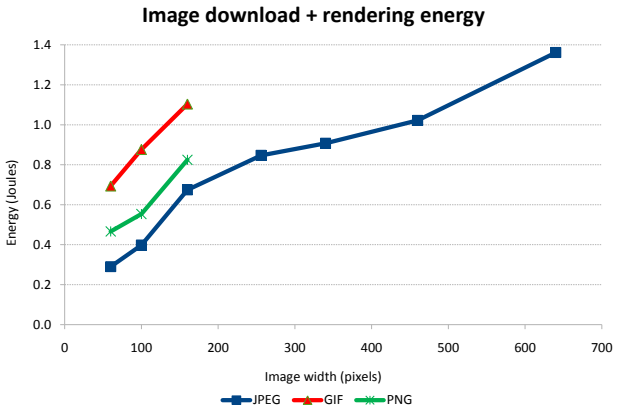


Figure 12: Energy consumption for image formats

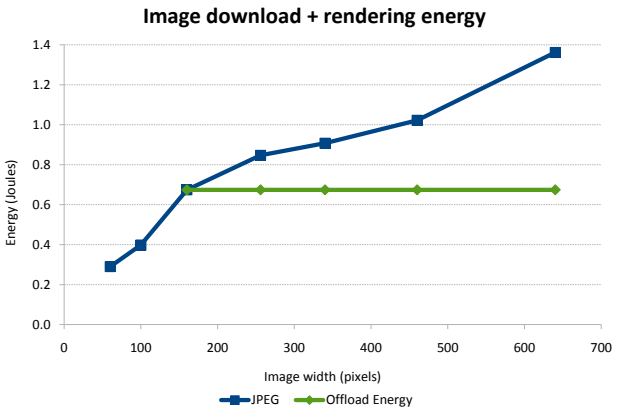


Figure 13: The benefits of offloading images

Figure 12 shows that JPEG is the most energy efficient format on the Android phone for *all* image sizes. To further drive this point home we used `Mogrify` to convert all images on the Amazon and Facebook pages to JPEG using a standard 92% quality compression measure. We then measured the energy consumption for rendering the resulting images from cache relative to the original images and obtained the following results

Site	Amazon (Joules)	Facebook (Joules)
Original	2.54	3.43
JPEG	2.04	2.39
Savings	**20%**	**30%**

The table shows that both Amazon and Facebook can conserve energy on Android phones by converting all their images to JPEG, without impacting the visible quality of the images. The reason for the savings is that JPEG compresses the images better and is faster to render then PNG and GIF.

5. OFFLOADING BROWSER COMPUTATION

Given the phone's limited energy there is a strong desire to minimize its work. A natural idea is to offload heavy computations to a server cloud and have the phone display the results [13]. In the context of web browsing one could offload image rendering — including decompression and conversion to a bitmap — to the cloud and have the phone simply display the resulting bitmap. Some browsers such as Opera [7] and SkyFire [6] take this approach. Their phone browsers talk to the web through a proxy that does most of the heavy lifting of rendering the page.

Generally speaking, there are two approaches to offloading computation:

- Front-end proxy: a web proxy examines all traffic to the phone and partially renders the page to save work for the phone. Here the proxy decides how the content should be modified before it is sent to the phone. This approach was previously used by old WAP gateways as they translated HTML to WAP. This approach is also used by Opera and SkyFire, but is not used by the default Android or iPhone browsers.
- Back-end server: the phone downloads web content as is, but then offloads certain operations to a server farm. Here the phone decides what needs to be offloaded.

In the next two section we discuss both approaches and measure their energy savings.

5.1 Offloading via a Front-end Proxy

Some sites, like `apple.com`, do not have a mobile version. Phones visiting these sites unnecessarily download large images. A natural application for a front-end proxy is to resize images to fit the phone screen, thereby saving radio use and rendering work. The natural place for a front-end proxy is at the carrier's data center where the carrier can optionally play the role of the proxy.

Figure 13 shows the energy savings that result from a front-end proxy that down-scales all large JPEG images to 160 pixel width. The blue line is the energy needed to download JPEG images of various sizes when no front-end proxy is used. This line is generated using the same cropping setup used in Figure 12. The green line shows the saving from a front-end proxy. Since all images larger than 160x160 are scaled down by the proxy, the energy beyond that image size is flat.

The area between the green line and the blue line represents the energy savings on the phone. On the Apple home page, for example, this front-end proxy would save 1.77 Joules on every page load with little impact to the user experience. While down scaling is not a new idea, we are not aware of any quantitative measurements showing its impact on modern phones such as the Android ADP2. We hope these results make a compelling case for this service.

Down-scaling is not without limitations:

- By down-scaling images, users loose the ability to quickly zoom in on intricate image details. Instead, a zoom-in requires downloading the zoomed part of the image. But the more common case is that no zooming takes place.
- For content sent encrypted using SSL, the proxy cannot see the content and therefore cannot down-scale it. However, down-scaling cleartext HTTP content is already a win.

Down-scaling can be offered as an opt-in/opt-out option to users to improve the browsing experience on the phone.

5.2 Offloading via a Back-end Server

Another option for offloading computation is to let the phone browser send sub-tasks to a back-end server. For images, for example, the cost of loading a compressed image and converting it to a bitmap can be offloaded. In this example we are potentially reducing CPU work at the cost of increasing use of the radio. We study when this trade-off is worth while.

As technology improves we expect to see the following trends:

- CPU energy consumption per instruction will continue to drop, and

Amazon images converted to:	Page size (kB)	Energy (mJ)
JPEG	26.45	224.68
PNG	65.53	268.05
Bitmap (BMP)	75.19	362.49

Table 2: Energy to render all the images in Amazon page in multiple formats

- The energy needed to transmit or receive one byte from the phone to the base station will stay roughly constant in comparison.

The first assumption is a qualitative version of Moore's law. The second assumption is due to the laws of physics: the energy needed to reach a base station at a certain distance is "roughly" constant. By "roughly" we mean that transmission energy will likely drop at a far slower pace than the rate of drop in CPU energy.

Given these two trends, it should be clear that offloading is *not* viable in the long run if it results in more radio use. In fact, as CPU energy per instruction decreases it is far better to maximize the amount of computation on the phone in order to minimize use of the radio.

Nevertheless, it is possible that with current technology offloading image rendering to a back-end server saves energy. If so, then one could envision an architecture where the phone sends an image URL to a back-end server. The back-end server retrieves the image, converts it to a plain bitmap, and sends the result to the phone. The phone simply copies the bitmap to its video buffer.

To test whether this architecture saves energy we compared the cost of fetching and rendering compressed JPEGs and PNGs to full bitmaps (BMP). We converted all the images on the Amazon page to one of JPEG, PNG or BMP. We then measured the cost of rendering the images on the page from cache for each of the three formats and the results are in Table 2.

As already suggested in Figure 12, JPEG is by far the most efficient encoding and PNG is the second. BMP is by far the worst even though it requires no decompression. We suspect the reason BMP does so poorly is that BMP images are considerably bigger and the extra energy needed to keep the radio on far outweighs the cost of decompressing the image.

Based on this experiment we conclude that back-end offloading of image rendering is not viable even with today's technology.

6. RELATED WORK

There is a large body of work focusing on energy consumption and network activity in mobile devices. Most results focus on the phone operating system or generic phone applications. To the best of our knowledge, none study the web browser and the energy needed to render specific pages.

Network traffic for smartphone applications. Existing research on mobile devices has proposed several approaches to the problem of minimizing energy consumption, such as [18] which reduces power consumption of data transfers, [10] which chooses wireless interfaces based on network condition estimation, [11] which proposes an approach to energy-aware cellular data scheduling, and [19, 9] which dynamically switches between wireless network interfaces based on the data traffic. Several techniques have been used (e.g., bundling multiple transfers [18], switching between WiFi and 3G cellular networks [19, 9], and scheduling based on a dynamic programming procedure for computing the optimal communication schedule [11]) to minimize energy consumption. In comparison, our focus is on helping web developers build more energy efficient web pages.

Other related measurement works include a study of the performance of 3G network and applications on smartphones [17, 23]. Huang et al. [17] show that 3G connections suffer from very long latencies and slow data transfers, which may lead to increased energy consumption. Zhuang et at. [23] present a location-sensing framework to improve the energy efficiency of localization on smartphones that run multiple location-based applications. The authors present four design principles that minimize energy, i.e., accelerometer-based suppression, location-sensing piggybacking, substitution of location-sensing mechanisms, and adaptation of sensing parameters when battery is low. Our work complements these works with different focus and methodology.

Partitioning applications. Prior works [16, 15, 20] investigated strategies for reducing the energy consumption of mobile phones by executing code remotely. Flinn et al. [16, 15] propose strategies on how to partition a program, how to handle state migration and adaptation of program partitioning scheme to changes in network conditions. Osman et al. [20] and Chun et al. [12] propose using full process or VM migration to allow remote execution without modifying the application code.

Cuervoy et al. [13] proposed a way to offload heavy computations to a server cloud and have the mobile phone display the results. In the context of web browsing, one could offload image rendering to the cloud and display the results back to the phone. While this works well for many applications, our experiments suggest that this approach does not improve browser efficiency. We showed that front-end offloading, as done by SkyFire [6] and Opera [7], can greatly reduce energy use on the phone.

Several previous studies [14, 21] also investigated the use of automatic program partitioning. Hunt et al. [14] develop strategies to automatic partitioning of DCOM applications into client and server components without modifying the application source code. Weinsberg et al. [22] propose an approach to offload computation to specialized disk controllers and processors (i.e., NICs).

7. SUMMARY AND DISCUSSION

While web pages are often optimized for speed and beauty, little attention is given to the amount of power needed to download and render the page.

We presented an experimental framework for measuring the power consumption of web pages, including specific components on the page. Our approach gives another dimension for evaluating mobile web sites and helps web developers build more energy efficient sites.

Designing energy-efficient web sites. Based on our experiments, we briefly summarize a few guidelines for designing energy-efficient web sites:

- Our experiments suggest that JPEG is the best image format for the Android browser and this holds for all image sizes.
- Gmail, the most "green" mobile site we found, uses HTML links to open email messages that the user clicks on. The desktop version of Gmail uses Javascript instead. Our experiments suggest that using links instead of Javascript greatly reduces the rendering energy for the page. Thus, by designing the mobile version of the site differently than its desktop version, Gmail was able to save energy on the phone.
- We found a number of static pages that could have been locally cached and displayed without any network access. Unfortunately, these sites link to Google Analytics, a tool that helps

monitor site usage. Javascript used by Google Analytics forces a dynamic network request that cannot be cached. Thus, even though the site could have been rendered from cache, the phone still has to pay the high cost of setting up a 3G session. We hope this paper will help web sites understand the cost of linking to these third party tools. Alternatively, if browsers exposed the state of the radio to Javascript then Google Analytics could choose not to report usage if the 3G radio is in low-power mode.

- AOL is able to save rendering energy by using a simple HTML table element to position elements on the page. Other sites that position elements using CSS need far more energy to render.
- On all the mobile sites we tested ads were small JPEG files and had little impact on overall power usage.
- Sites like `apple.com` are particularly energy hungry. We hope this paper demonstrates the importance of building a mobile site optimized for mobile devices. Sites who do not, end up draining the battery of visiting phones. This can potentially reduce traffic to the site.

Future work. Our experiments focused on the energy consumption of specific pages with the goal of improving the energy signature of those pages. It would be interesting to extend these results and study the energy signature of an entire browsing session at a site where the user moves from page to page at that site. During the session, web elements such as CSS and images will be cached locally. Therefore, we cannot estimate the energy cost of a session by simply summing the energies of pages visited during the session. Measuring an entire typical session may help optimize the power signature of the entire web site.

Another interesting direction is to more accurately model the energy consumption of the 3G radio as the browser fetches web pages. Our model from Section 2.5 works well for web pages with a limited number of components, but breaks down for other pages. We conjecture that a detailed understanding of the shape of the traffic generated by the browser will be needed to estimate the energy used by the radio. Understanding how the radio is used when the browser fetches a web page could help browser vendors optimize the browser's multi-threaded system for downloading pages.

Acknowledgments

The fourth author was supported by Deutsche Telekom and NSF.

8. REFERENCES

[1] Agilent 34410A Digital Multimeter. `http://www.home.agilent.com/agilent/product.jspx?pn=34410A`.

[2] Android Developer Phone 2 (ADP2). `http://developer.htc.com/google-io-device.html`.

[3] Android Developers - Activity. `http://developer.android.com/reference/android/app/Activity.html`.

[4] Android Developers - Intents. `http://developer.android.com/reference/android/content/Intent.html`.

[5] NetMarketShare. `http://www.netmarketshare.com/report.aspx?qprid=61&sample=37`.

[6] SkyFire. `http://skyfire.com`.

[7] The Opera browser. `http://opera.com`.

[8] WebKit. `http://webkit.org`.

[9] A. Rahmati, C. Shepard, A. Nicoara, L. Zhong, J. Singh. Mobile TCP Usage Characteristics and the Feasibility of Network Migration without Infrastructure Support. In *Proc. of ACM 16th International Conference on Mobile Computing and Networking (MobiCom'10), Chicago, Illinois, USA*, 2010.

[10] A. Rahmati, L. Zhong. Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer. In *Proc. of ACM 5th International Conference on Mobile Systems, Applications, and Services (MobiSys'07), Puerto Rico*, 2007.

[11] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, V. N. Padmanabhan. Bartendr: A Practical Approach to Energy-aware Cellular Data Scheduling. In *Proc. of ACM 16th Annual International Conference on Mobile Computing and Networking (MobiCom'10), Chicago, USA*, 2010.

[12] Byung-Gon Chun, Petros Maniatis. Augmented Smartphone Applications Through Clone Cloud Execution. In *Proc. of the 12th Conference on Hot Topics in Operating Systems*, 2009.

[13] E. Cuervoy, A. Balasubramanian, D-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. of ACM 8th Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys'10), San Francisco, USA*, 2010.

[14] G. C. Hunt, M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99), Louisiana*, 1999.

[15] J. Flinn, D. Narayanan, M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS), Germany*, 2001.

[16] J. Flinn, S. Park, M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria*, 2002.

[17] J. Huang, Q. Xu, B. Tiwana, A. Wolman, Z. M. Mao, M. Zhang, P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *Proc. of ACM 8th Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys'10), San Francisco, USA*, 2010.

[18] N. Balasubramanian, A. Balasubramanian, A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. of ACM SIGCOMM Internet Measurement Conference (IMC'09), Chicago, USA*, 2009.

[19] S. Nirjon, A. Nicoara, C. Hsu, J. Singh, J. Stankovic. MultiNets: Policy Oriented Real-Time Switching of Wireless Interfaces on Mobile Devices. In *Proc. of 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12), China*, 2012.

[20] S. Osman, D. Subhraveti, G. Su, J. Nieh. The Design and Implementation of Zap. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, USA*, 2002.

[21] U. Kremer, J. Hicks, J. M. Rehg. Compiler-Directed Remote Task Execution for Power Management. In *Proc. of The Workshop on Compilers and Operating Systems for Low Power (COLP)*, 2000.

[22] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, P. Wyckoff. Tapping into the Fountain of CPUs - On Operating System Support for Programmable Devices. In *Proc. of the 13th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), Seattle, WA, USA*, 2008.

[23] Z. Zhuang, K. Kim, J. Singh. Improving Energy Efficiency of Location Sensing on Smartphones. In *Proc. of ACM 8th Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys'10), San Francisco, USA*, 2010.