Cloud is not a silver bullet: A Case Study of Cloud-based Mobile Browsing

Ashiwan Sivakumar[†], Vijay Gopalakrishnan[‡], Seungjoon Lee [‡], Sanjay Rao[†],

Subhabrata Sen[‡] and Oliver Spatscheck[‡]

[†] Purdue University [‡] AT&T Labs - Research

ABSTRACT

In recent years, there has been growing interest in both industry and academia in augmenting mobile web browsing with support from the cloud [4, 1, 3, 16, 18]). These efforts are motivated by the goals of lowering costs of data transfer, and reducing web latencies and device energy consumption. While these efforts have adopted different approaches to cloud-based browsing, there isn't a systematic understanding of the rich design space due to the proprietary nature of many of the solutions. In this paper, we take a step towards obtaining a better understanding by evaluating an extreme point in the design space that involves cloud support for most browsing functionality including execution of *JavaScript (JS)*, and for compaction of data (e.g., image transcoding and compression). Our study is conducted in the context of *Cloud Browser (CB)*, a popular commercially available browser that embodies this design point. Our results indicate that *CB* does not provide clear benefits over *Direct* (*a device-based browser*) either in energy or download time. For e.g. while *CB* decreases the download time compared to *Direct* for 38.87% of pages, it increases it by as much as 29.8s for other pages. Similarly while *CB* decreases the total energy by up to 20.77J compared to *Direct* for 52.7% of the pages, it increases it by up to 21.31J for other pages. Interestingly, even though *CB* does *JS* execution in the cloud, it increases the CPU and network energy for close to 50% of the pages. Overall our study indicates that cloud-based browsing is not always a win, and there are important trade-offs that must be carefully considered when moving functionality to the cloud.

1 Introduction

Mobile internet users are growing rapidly given the spread of higher speed cellular technologies like 3G and LTE. By 2014, it is predicted that mobile internet usage will surpass desktop internet usage [2]. There has been much interest in using cellular-connected devices such as smart-phones to perform network-related tasks and the market has responded with a wide variety of applications that serve this growing need.

In recent years, there has been much interest in both academia (e.g., [16], [18], [12]) and industry (e.g., [4], [1], [3], [5]) in using the cloud to augment mobile web browsing, in order to overcome the processing and energy limitations of mobile devices. These efforts are related to but distinct from efforts like [15, 8, 10, 7, 11, 14] which develop frameworks for offloading code of applications (e.g., face recognition) that primarily run on the mobile device to the cloud. In contrast, in mobile web browsing, data naturally flows into mobile devices from remote servers, and cloud servers could potentially be on the data path from the server to the device. The potential benefits with cloud-based mobile browsing include improving data download time, reducing device energy consumption, and reducing data usage and costs. The technology has sufficiently matured that there are a number of cloud-based mobile web browsers that are available in the market – popular ones include Opera Mini [4], Amazon Silk [1], Sky Fire [5] and Chrome beta [3].

The existing approaches to cloud-based web browsing represent a range of points in a rich design space. At one end of the design spectrum we have the traditional mobile browser that performs all browsing functionality in the client. At the other end of the spectrum, there are many cloud browsers that take a 'cloud-heavy thin-client' approach (e.g., [18, 4, 5]) relying on cloud support for most functionality including parsing and rendering web pages, *JavaScript (JS)* execution, and compaction of data (e.g., data compression, transcoding images). Other approaches move a subset of browser functionality to the cloud – e.g., [3] executes *JS* at the client but uses the cloud to fetch individual objects and perform data compaction, while [16] has argued for offloading parts of the page load process, though a specific design is not provided. Overall there is limited understanding in the community today of the trade-offs between these different design points.

In this paper we take a first step towards understanding the performance implications of mobile cloud browsing solutions by comparing the two extreme points – one that does not use the cloud at all, and another that primarily relies on the cloud. Our evaluation is conducted in the context of a popular commercially available cloud-based mobile browser (as per recent reports it has about 300 million unique users), that uses the cloud for performing *JS* execution and data compaction. To keep the focus on the scientific aspects of our study, we anonymize the browser and call it *Cloud Browser (CB)*. We compare *CB* with a traditional browser that runs locally in the device (which we refer to as *Direct*).

ACM HotMobile'14, February 26–27, 2014, Santa Barbara, CA, USA.

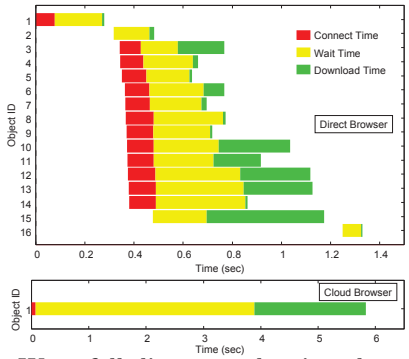

Figure 1: Waterfall diagrams showing the network activity for a page download with *Direct* and *CB*.

Our evaluation focuses on two metrics: (i) page download time, and (ii) device energy consumption. While the former directly impacts user experience, the latter captures the energy consumed by both the CPU and the network (the power consumed by the radio interface is known to contribute a considerable fraction of the total device power [9]). We consider this metric to be relevant for the foreseeable future since even though processing capabilities of mobile devices have dramatically improved in recent years, battery capacity remains a major resource limitation.

Our evaluations indicate that neither *Direct* nor *CB* is better under all scenarios. For e.g. while *CB* decreases the download time compared to *Direct* for 38.87% of pages, it increases it by as much as 29.8s for other pages. Similarly *CB* increases energy usage by up to 21.31J compared to *Direct* for some pages. While *CB* saves CPU energy by offloading *JS* to the cloud and network energy by not sending the *JS* to the client, interestingly it does worse for many pages that are heavy on *JS* processing. Further, the benefits of running *JS* in the cloud are less clear over user sessions that involve extensive client interactivity.

We also isolate the impact of data compaction through experiments with pages without *JS* content. We find that for such pages, *CB* does better than *Direct* in total energy only for 17% and increases the network energy usage for 75% of the pages. Data compaction does not always lead to network energy savings due to the time involved in performing these sophisticated data compaction tasks, and the complex network radio state transitions.

Our contributions in this paper are (i) we conduct one of the first studies of operational cloud-based mobile browsers - such a study is useful in its own right given the limited technical information available on many existing solutions today; and (ii) we take a first step towards understanding the trade-offs involved in moving browser functionality, specifically *JS* execution, and data compaction, to the cloud. While our evaluations are conducted with *CB*, we believe the results expose important and more broadly applicable trade-offs that must be considered when using the cloud to support these functionalities.

2 Background

In this section we give an overview of the working of *CB* based on a combination of publicly available information and carefully constructed experiments.

Figure 1 shows the waterfall diagram for network activity when downloading a page using *Direct* and *CB*. Each bar corresponds to a HTTP object request-response pair. We break the total time for each bar into three parts - **1. Setup time, 2. Wait time, 3. Download Time**. The Setup time includes the times for DNS resolution, initial connection establishment and the actual URL request; The *Wait Time* is the time before the client starts receiving the data; The Download time is the time taken to transfer the data from the server to the client. The *Wait Time* in *CB* consists of (i) processing time at the cloud proxy like parsing, processing *JS* etc. and (ii) time taken by the proxy to fetch all the objects from the server.

As shown in the figure, *Direct* first fetches the main HTML page directly from the web server, parses it and loads the other objects required to render the page using multiple (parallel) HTTP connections. The page is then rendered locally on the device. On the other hand, *CB* connects to the cloud proxy and after a *Wait Time* receives a compact version of the page from the proxy in a proprietary format (which we call *Cloud Browser Markup Language (CBML)*). During the *Wait Time*, the cloud proxy employs data compaction techniques on the page like reformatting, re-sizing images, compression for small-screen rendering. The client extracts the *CBML* and renders the page.

CB offloads *JS* processing to the cloud proxy. It supports two modes of operation for pages with *JS*. In older versions *CB* supports a mode where all *JS* in a page are run for a timeout of 5s, after which they are stopped and a *CBML* is sent to the client. There are many web pages that change their content either automatically (e.g., timer-based) or through user-interaction (e.g., user-clicks) using *JS*. The older version could not accurately render many pages that change content through long-running *JS* ($>$ 5s) and does not support rich interactivity. In newer versions *CB* supports a second mode where the client opens a secure, persistent connection with the cloud proxy and the proxy continuously runs the *JS*, pushes different bursts of objects at different times whenever the page changes. This approach helps *CB* accurately render most pages. Even with the second approach, in our experiments we have seen *CB* failing to render some pages that continuously download objects through *JS* that run forever. We use the second mode of *CB* in our experiments since it renders most pages accurately.

3 Evaluating Cloud Browser Design

We evaluate key design decisions taken by existing mobile cloud browsers in the context of *CB*. In this section we describe the evaluation goals and methodology.

Evaluation goals: In evaluating *CB*, our primary goals are (i) to understand and quantify the impact of offloading *JS* execution to the cloud under various scenarios and (ii) to understand the benefits of performing data compaction of pages in the cloud.

Setup and Methodology: We ran experiments using *CB* and *Direct* on a Samsung galaxy S3 phone using an LTE network in West Lafayette. We choose 40 of the top US pages in Alexa [6]. The pages cover a wide range of categories like news, sports, photo streaming, business and science. We conduct experiments in the wild by downloading these pages using *Direct* and *CB*. We conduct 20 experiment runs with each page. To subject the two schemes to similar signal strength conditions, in each run we first download a page using *Direct* and then using *CB* back-to-back. Moreover, we request the URLs with 60 secs time interval between each request when using both *Direct* and *CB*. We run

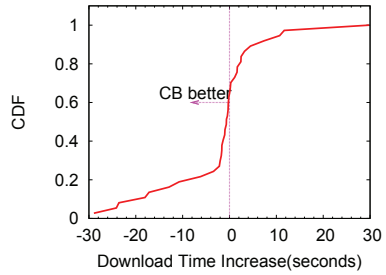

Figure 2: CDF of the median *Download time increase (Time CB - Time Dir)* for each page with *CB*.

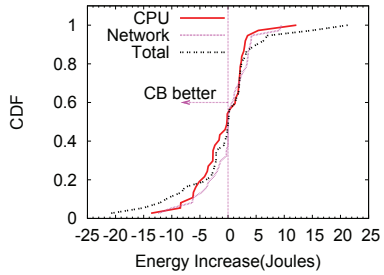

Figure 3: CDF of the median *Energy increase (Energy CB - Energy Dir)* for each page with *CB*.

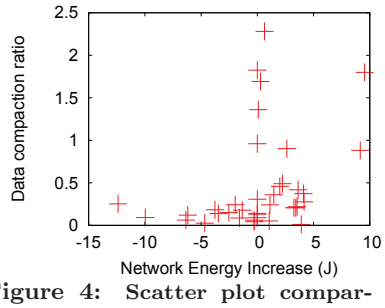

Figure 4: Scatter plot comparing the data compaction ratio and the median *increase in energy* (network) for each page with *CB*.

a majority of our experiments during the night time when the load in the cell tower is low. Since we observe intermittent cellular technology hand-off (LTE to 3G), we log the network type throughout the experiment in a background process and filter out runs that do not use LTE completely for all schemes. We have at least 15 runs per page after this filtering step. Further we collect packet traces on the device to analyze request-response timings, object sizes, and other TCP flow-level information for each page download.

Most of our experiments focus on the first-time download of a web page, and hence we disable local device caching for both *CB* and *Direct* unless otherwise mentioned. In practice, we believe enabling local caching on the device will benefit *Direct* more since (i) caching of *JS* and CSS files helps *Direct* but not *CB*, since the latter processes these files in the proxy and does not send them to the client; and (ii) *CB* does not use a persistent cache – consequently, cached objects disappear between invocations of the application. Finally, disabling caching ensures that our comparison results are not impacted by proprietary implementation artifacts of *CB* (e.g., how effectively the proxy learns the content of the client cache). Our experiments in Section 4.2 do however consider local caching for both schemes since the focus of these experiments is on performance over an entire user session, and not first-time page downloads alone.

Metrics Used: We consider two important metrics:

• *Page download time:* We define page download time as the time between the first SYN and the last ACK for all objects in a page, as observed in the packet traces collected on the device. We measured (i) the onload time, i.e., the time until an onload event occurs at the browser. An onload event denotes the browser has finished loading the web-page and this includes time to run a subset of *JS* (e.g., synchronous *JS*); and (ii) the page download time, as described above. For all the pages we considered, the browser onload event was found to be smaller than the page download time. Thus, with *Direct*, the page download time metric includes all *JS* execution until the onload event, as well as *JS* execution time after the onload until the last object downloaded. While the metric does not include the *JS* execution time beyond the last object downloaded, this time is observed to be relatively small for the pages we considered.

• *Total energy:* This is the aggregate device energy consumption and consists of two components: (i) CPU energy consumed by a browser process to download a web page and render it on the device. We collected the CPU utilization of the *Direct* and *CB* processes (every 100 ms) for the full 60 sec period from when the download was initiated. We found this to be a sufficiently long duration for all *JS* to complete execution for the pages we considered. We then calculate the CPU energy by using the PowerTutor model [17] whose input is the CPU utilization collected as described above. (ii) Network energy (communication) consumed by the radio interface for a web page download. We calculate the network energy value using the open source ARO tool [13], which captures the Radio Resource Control (RRC) state transitions [1]and the corresponding energy consumption levels by performing fine-grained simulation on the packet traces collected from the client device (UE) We focus on CPU and network energy since they account for the bulk of the difference in device energy consumption between *CB* and *Direct*. While we do not consider screen energy in this work, we believe our download time metric should correlate to the duration for which the screen is on. However, incorporating screen energy is a direction of future work.

We preferred using energy models to measure the CPU and network energy consumption, than a direct power monitor hardware because running experiments using a power monitor requires fair bit of manual intervention to accurately collect the traces discarding noisy components. Since we were running a large number of experiments (several pages, with at least 20 runs for each scheme and each page), we wanted to make it fully automated for better scalability of experiment runs. However, we have validated our models with measurements using power monitor hardware.

4 Results

4.1 Impact of offloading JavaScript execution

In this section we present results from experiments in the wild using the setup described in section 3. Since many

[1]In 3G/LTE networks, a key factor affecting the network energy efficiency is the Radio Resource Control (RRC) state machine which is designed to efficiently utilize limited radio resources and to improve the device battery life time. The LTE spec defines two states - RRC_CONNECTED (the radio is active and consumes data) and RRC_IDLE (the radio is idle and promotes to RRC_CONNECTED before consuming data) [9]. Further, each of these states implements a mechanism to save power without impacting latency called Discontinuous Reception (DRX), with available/unavailable RX cycles/periods. In RRC_CONNECTED state, the device will be in a higher power Continuous Reception (CR) mode when actually consuming data and in a lower power LONG_DRX mode otherwise. The power consumed in LONG_DRX is higher than that in the RRC_IDLE state.

modern web pages have dynamic content generated by *JS*, we choose 40 pages from the top 100 Alexa pages, all containing *JS* for the experiments. In order to reduce the impact of signal strength variability on the experiment results, we perform multiple runs for each of the pages back-to-back, first using *Direct* and then using *CB* (as described in section 3). We then compute the *Energy increase (Energy CB - Energy Dir)* and *Download time increase (Time CB - Time Dir)* for each of the back-to-back runs. In figure 2 we present the median of the *increase in download time* and in Figure 3 the median of the *increase in energy* for each page including CPU, communication (network) and total.

We observe from figures 2 and 3 that *CB* does not provide clear benefits over *Direct* either in energy or download time. e.g., while *CB* decreases the download time compared to *Direct* for 38.87% of pages, it increases the download time by as much as 29.8s for other pages. Similarly while *CB* decreases the total energy by up to 20.77J compared to *Direct* for 52.7% of the pages, it increases the total energy by up to 21.31J for other pages. Interestingly, even though *CB* does *JS* execution in the cloud, it increases the CPU and network energy for close to 50% of the pages.

On further analysis, we found a few key factors that determine when *CB* is beneficial:

Extent of JS *in the page:* In general, for pages that are heavy on *JS* processing (CPU intensive), *CB* saves on CPU energy as the *JS* is processed in the cloud while *Direct* processes *JS* locally on the device. Further, *CB* saves on network energy by not shipping the *JS* code down to the client, thereby achieving significant data compaction (atleast 61%) on such pages. For pages that are light on *JS* processing, generally *CB* does not save on CPU energy because the overhead involved in decompressing the *CBML* (the proprietary format of the page sent by the *CB* proxy) outweighs the savings obtained by not processing the *JS* locally, and the savings in network energy may be small because *CB* does not decrease the bytes transferred considerably.

Long Vs. short running JS*:* While the previous observation may be expected, interestingly we find many pages that are heavy on *JS*, but using *Direct* is better than *CB*. To aid our understanding on why *Direct* does better for such pages, we choose two pages, one where *CB* does better than *Direct* and other where *CB* does worse and look at the RRC state transitions for one download for both the pages in figure 5.

As seen from figure 5(a) for 'page A', *CB* transmits all the data in one burst. Further, we found that *CB* transmits a single burst of data for pages that complete all *JS* execution in a short time ($<$ 5s). For 'page B', as shown in figure 5(b) *CB* sends multiple bursts of data at different times. This page has a long-running *JS* ($>$ 10s) that runs towards the end of the page load and downloads 5 different images one after another with a fixed timeout, then shows each image in the form of a slide show. In newer versions *CB* supports long-running *JS* by streaming different bursts of data to the client (Section 2) and hence we see multiple bursts of data for this page.

To further understand why streaming multiple bursts from the cloud can potentially hurt network energy, we show a scatter plot comparing the data compaction ratio for each page with the corresponding median *increase in energy* (network) in figure 4. It can be observed from the figure that in all pages where *CB* does better than *Direct*, it achieves significant data compaction whereas for many pages where *CB*

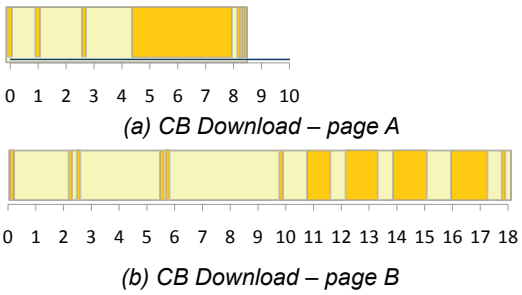

Figure 5: LTE RRC state transition diagram for two page downloads with *CB*. *CB* does better in 'page A' and *Direct* does better in 'page B'. The periods with active data transfer are shown as darker regions (dark yellow) and the inactive periods are shown with lighter regions (light yellow). Note that the radio is still on during the inactive periods too [9].

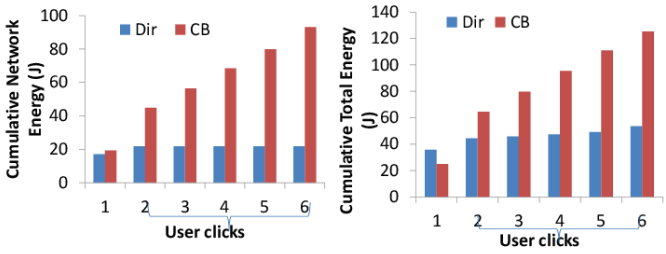

Figure 6: Running *JS* in the cloud impacts network energy with user interactivity in a session

does worse, it obtains lesser compaction. In fact for some pages where *CB* does worse, the compaction ratio is $>$1. Further we found that for pages where *CB* streams multiple bursts of data it is not able to achieve better compaction when compared to pages where it compresses the whole page and sends in one burst. A possible reason is each of the bursts might have redundant data since *CB* streams multiple *CBML* for the page. We hypothesize, sending deltas in each burst could potentially help achieve better data compaction. Using multiple bursts may diminish network energy savings (since the radio may need to be in the connected state throughout).

Overall these results indicate that even though offloading *JS* to the cloud could potentially reduce CPU energy, the benefits have to be carefully weighed against the different trade-offs based on the page and the network characteristics.

4.2 Comparing energy usage over a user session

In the previous section, we focused on one-time download of a web page, whereas in practice, users may interact with pages a lot. e.g., users may submit forms, click on images or hyperlinks etc. If *JS* processing is completely offloaded to the cloud, any interactivity may involve communicating with the cloud proxy throughout the session. So in this section we compare the cumulative energy usage with *CB* and *Direct* over an entire user session to study the impact of user interactivity.

Here we choose 'page B' from the figure 5. As described earlier, page B shows a slide show of 5 images and contains buttons for the user to view each images separately. We con-

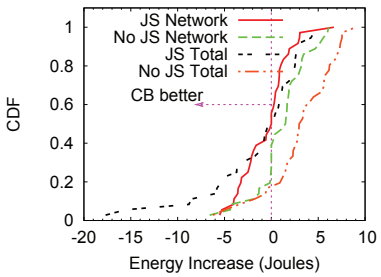

Figure 7: CDF of the median *increase in energy* (network, total) for pages with and without *JS*

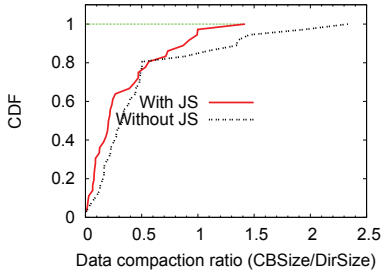

Figure 8: CDF of data compaction ratios - ratio of *CB* size and *Direct* size for all pages with and without *JS*

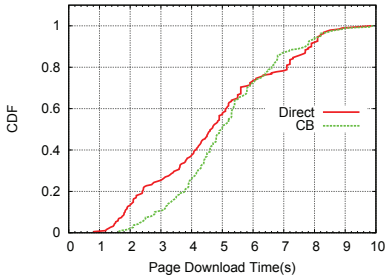

Figure 9: CDF of page download times using *CB* and *Direct* using a 3G network in NJ

duct an experiment, where we emulate an interactive session by generating *user clicks* on the buttons once every minute to view a particular image. Since the focus of this experiment is on a single user session, we enable caching on the device for both *Direct* and *CB*. Each impulse in figure 6 from minutes 2 to 6 represents a user click.

Figure 6 shows the cumulative energy usage (total and network) every minute for the whole session. When the user clicks on the button, *Direct* runs the *JS* locally in the client and displays the particular image that is already cached during the first download of the page. Since all 5 images are available in the local cache, *Direct* needs no further data transfer and does not consume any network energy. This is easily seen from figure 6 as the network energy with *Direct* remains unchanged for the whole session, while the total energy increases by as much as 9.2J over the whole session, attributed to the CPU energy consumed by running the *JS* locally.

In contrast, since *CB* does not send the *JS* to the client and processes it in the cloud, every user click results in the client communicating with the cloud proxy to run the *JS* and send a new *CBML* of the page to the client. Despite performing data compaction on each of these different *CBML*, *CB* increases both the cumulative network and total energy over the session. e.g., while the cumulative CPU energy consumed by *CB* is almost the same as *Direct* at the end of the session, it consumes about 60.9J more network energy than *Direct* for the whole session, thus consuming considerably more total energy overall. We conclude that the benefits with *CB* are not clear if there is lot of user interactivity because offloading *JS* to the cloud increases the communication cost significantly over a user session.

4.3 Impact of data compaction

There are two key design elements that could impact energy savings with *CB*: (i) offloading *JS* execution; and (ii) data compaction. In this section we separate these two factors and study the impact of data compaction in greater detail. Data compaction here refers to techniques that reduce data usage by transforming the web page (e.g., reformatting, resizing images for small form factor etc.) and excludes the reduction obtained by not shipping the *JS* down to the client. We note that the data compaction libraries used by *CB* are well-known and many mobile browsers adopt these libraries.

For this study, we prepare and host snapshots of the 40 pages on our local server in Purdue to understand the impact of data compaction in a more controlled setting. We then remove all *JS* code from these pages thus preparing two versions of each page - one with *JS* and other without. We conduct an experiment in which we first download the *JS* version of a page using both *CB* and *Direct* back-to-back, then download the version without *JS* using both *CB* and *Direct*. Similar to the experiments in section 4.1, we perform 20 runs for each version of the page and compute the *Energy increase (Energy CB - Energy Dir)* for both the versions of the pages.

In figure 7 we present the median *increase in energy* (both network and total) with *CB* compared to *Direct* for both versions - one with *JS* and another without *JS*. The figure shows that *CB* does not always perform better than *Direct* for both versions of the pages (with and without *JS*). Moreover, the benefits with *CB* are more limited for pages without *JS*. than pages with *JS*. For instance, *CB* increases the total energy for 83% of pages without *JS* as compared to only 47.3% for the *JS* version of the same pages. To further aid our analysis, we plot the data compaction ratio for the pages with and without *JS* in figure 8. We see that the data compaction ratios are smaller when *JS* is excluded from the pages, explaining the higher increase in energy with *CB* when compared to the version with *JS*. As a side point, it is interesting to note that there are some pages for which the data compaction ratio is > 1, even when *CB* downloads these pages in a single burst. We hypothesize that *CB* might send page layout information in the *CBML* to aid the page rendering in the client and this might be an additional overhead for small pages.

To understand the correlation between the data compaction ratio and the energy savings with *CB*, we analyzed the data further and found a few factors that determine when data compaction is beneficial. In general as expected, while the energy savings with *CB* does show some correlation with the compaction ratio, we also find the *Wait Time* at the proxy (Section 2) to be a key reason for the additional energy expended by *CB*. When the *Wait Time* is large, the total download time increases. For the pages without *JS*, *CB* incurs higher download time for 60% of pages. e.g., we find a page for which *CB* consumes more network energy than *Direct*, even though the data compaction ratio achieved by *CB* is 0.09. This is because, the *CB* proxy takes a long *Wait Time* (6.8s) to download the page from the server and perform data compaction, thereby increasing the total download time compared to *Direct*.

A higher download time leads to increase in network energy since the client radio stays in the RRC_CONNECTED state [9] throughout the download. In this state, the client will be in LONG_DRX mode periodically, consuming relatively lower power than the CR state (refer footnote [1]). However spending a longer time in LONG_DRX through higher *Wait Time*, can outweigh the energy savings obtained by reducing the time spent in the CR state by compression. Thus our results suggest that the decision to perform data compaction, should take *Wait Time* and the LTE radio state transitions into account.

4.4 Sensitivity study

In this section we perform a sensitivity study using a 3G network in NJ and a tablet device (Samsung galaxy tab 8.9). We download the *non-JS* version of the pages that are hosted from our server in Purdue locally. Again we find *CB* does not provide clear benefits over *Direct*. e.g., Figure 9 shows *CB* increases the page download time compared to *Direct* for 70% of pages. We also observed that *CB* increases the total energy compared to *Direct* for about 60% of the pages. Specifically, *CB* increases the network energy by up to 6J for 40% of the pages and increases the CPU energy for all the pages compared to *Direct*. Since these pages do not have *JS*, it may be that *CB* consumes more CPU cycles to decompress the *CBML* thereby consuming more CPU energy than *Direct*. We omit the energy results due to lack of space.

5 Conclusions

In this paper, we argue that there is need to revisit trade-offs in the design of cloud-assisted mobile browsing given the dramatic improvements in the processing capabilities of mobile devices, and given that the cellular radio interface constitutes a growing component of the total device power. We substantiate these arguments through one of the first studies of an operational and widely used cloud-based mobile browsing solution.

In particular, our observations are:

•*Offloading JS to the cloud is not necessarily beneficial:* While the conventional wisdom is that offloading *JS* to the cloud could lead to much benefits, our evaluations indicate the benefits are dependent on the the characteristics of pages – e.g., by offloading *JS* to the cloud, CB decreased the total energy for 52.7% of the pages while increasing the total energy by up to 21.31J for remaining pages. While pages heavy on *JS* processing do better with cloud-based solutions, pages with long-running *JS* involving periodic data downloads from servers (e.g., to support rolling advertisements) are harder to support in an energy efficient manner for cloud-based solutions. For such pages, transmitting all data in a single burst may reduce response times for the client – use of multiple bursts may diminish network energy savings (since the radio may need to be in the connected state throughout), and lead to less effective data compaction. Performing page characteristic analysis to determine the best design for a given page is an important direction for future research.

•*Considering user interactivity when offloading JS is important:* Supporting interactive sessions in a energy-efficient manner is challenging when offloading *JS* to the cloud. The network energy consumed owing to communication with the cloud server for each client interaction may outweigh the CPU energy saved by processing *JS* in the cloud – e.g., in an interactive session experiment, *CB* increased the overall network energy by as much as 60.9J compared to *Direct*.

•*Data compaction is not always beneficial:* Performing data compaction in the cloud is not always beneficial and the benefits in energy savings have to be weighed against the time taken to perform data compaction, in order to not increase the network energy – e.g., in our experiments with pages without *JS* content, *CB* performed better than *Direct* only for 25% of the pages in network energy and 17% in total energy. Even though *CB* achieved 90% compaction of data on some pages, it increased the energy usage by as much as 9.8J because of the time to perform the compaction.

Overall, our study points to the need to carefully consider these new trade-offs while designing cloud-based mobile browsing solutions. As part of our ongoing work, we are investigating other design points that have been taken by existing cloud-based mobile browsers, as well as designing new solutions driven by our insights.

6 Acknowledgments

We thank our shepherd Michael Piatek and the anonymous reviewers for their constructive feedback and comments. This work was supported in part by National Science Foundation (NSF) Career Award No. 0953622. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

7 References

[1] Amazon silk split browser architecture. `https://s3.amazonaws.com/awsdocs/AmazonSilk/latest/silk-dg.pdf`.

[2] By 2014, mobile internet usage will take over desktop internet usage. `http://www.digitalbuzzblog.com/2011-mobile-statistics-stats-facts-marketing-infographic/`.

[3] Data compression proxy in android chrome beta. `https://developers.google.com/chrome/mobile/docs/data-compression`.

[4] Opera mini architecture and javascript. `http://dev.opera.com/articles/view/opera-mini-and-javascript/`.

[5] Skyfire - cloud based mobile optimization browser. `http://www.skyfire.com/operator-solutions/whitepapers`.

[6] Alexa. Available at `http://www.alexa.com/topsites`.

[7] B.-G. Chun et al. Clonecloud: Elastic execution between mobile device and cloud. In *Proc. ACM Eurosys*, 2011.

[8] E. Cuervo et al. Maui:making smartphones last longer with code offload. In *Proc. ACM MobiSys*, 2010.

[9] J. Huang et al. A close examination of performance and power characteristics of 4g lte networks. In *Proc. ACM Mobisys*, 2012.

[10] R. Kemp et al. Cuckoo: a computation offloading framework for smartphones. In *Proc. MobiCASE*, 2010.

[11] S. Kosta et al. Thinkair: Dynamic resource allocation and parallel execution in cloud for mobile code offloading. In *Proc. IEEE INFOCOM*, 2012.

[12] K. Matsudaira. Making the mobile web faster. *Communications of the ACM, Vol 56. No 3.*, 2013.

[13] F. Qian et al. Profiling resource usage for mobile applications: A cross-layer approach. In *Proc. ACM Mobisys*, 2011.

[14] A. Saarinen et al. Can offloading save energy for popular apps. In *Proc. ACM MobiArch*, 2012.

[15] M. Satyanarayanan et al. The case for vm-based cloudlets in mobile computing. *IEEE/Trans. Pervasive Computing*, 2009.

[16] X. S. Wang et al. Accelerating the mobile web with selective offloading. In *Proc. ACM MCC*, 2013.

[17] L. Zhang et al. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. ACM CODES+ ISSS*, 2010.

[18] B. Zhao et al. Reducing the delay and power consumption of web browsing on smartphones in 3g networks. In *Proc. ICDCS*, 2011.