Accelerating the Mobile Web with Selective Offloading

Xiao Sophia Wang
University of Washington
Seattle, Washington, USA
wangxiao@cs.washington.edu

Haichen Shen
University of Washington
Seattle, Washington, USA
haichen@cs.washington.edu

David Wetherall
University of Washington
Seattle, Washington, USA
djw@cs.washington.edu

ABSTRACT

Mobile Web page loads are notoriously slow due to limited computing power and slow network access. Our preliminary experiments show that computation is a significant fraction of page load time on mobile devices. Also, energy arguments suggest that it will stay this way. To compensate the limited computing power, our position is that offloading portions of the page load process to the cloud can significantly improve mobile page load time. We propose a measurement-based framework that allows to offload portions of mobile page load process to the cloud. Unlike browsers that offload fixed parts of page loads such as Opera Mini, our framework will allow to offload *any* portion of the page load process. We will experiment with a large variety of real-world situations (e.g., varying computing power on mobile devices) by offloading varying portions of page loads using our framework. Informed by the experimental results, we will develop a mobile browser that considers the diverse situations as well as energy and data usage.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design

Keywords

Web; Mobile Web; Page load; Cloud; Offload

1. INTRODUCTION

Web pages delivered by HTTP(S) have become the defacto standard for connecting billions of users to Internet applications. While the Web provides many favorable features (e.g., mashups) as opposed to native applications, it incurs noticeably high page load latencies on mobile devices, making it less attractive on mobile. Our preliminary tests on the top ten mobile pages suggest that mobile page load times are 1.5x (3x) of desktop counterparts in initial (repeated) loads.

Our previous research in desktop settings that quantifies the composition of page load time suggests that computation is significant [15]. We extensively loaded the top 200 Alexa [1] Web pages and found that computation comprises 35% of page load latencies on the critical path (the longest bottleneck path) even on an iMac with a 3GHz quad core CPU and a 8GB memory. This fraction is as much as 40% on a machine with 2GHz CPU and a 4GB memory. Our mobile study on sampled Web pages also suggests that computation is a key bottleneck of Web page loads on mobile devices that have less computing power than desktop. And it is likely that mobile computing power will stay this way in the future due to limited energy on mobile devices.

To compensate the limited computing power on mobile devices, a common approach is offloading to the cloud. However, little has been done to offload mobile Web page loads in the research community because it requires deep understanding of how browsers load Web pages. Offloading is now adopted in more browsers in the industry such as Opera Mini [11], Amazon Silk [12], and recently released Android Chrome Beta [2]. Those browsers do offload to the cloud, but it is unclear how well they perform. Opera Mini compresses pages to a markup format called OBML that reduces both transfer time and data usage [10]. Yet it still leaves the full logic of page load processing to mobile devices and so does Android Chrome Beta. Amazon Silk moves portions of page loads to the cloud which depend on the structure of Web pages. But, it is unknown how the Silk browser divides the page load process and the choice of offloaded portions should consider ambient situations, not only Web pages.

Our position is that offloading portions of the page load process to the cloud can significantly accelerate the mobile Web. We want a mobile browser that selectively offloads portions of page loads in any real-world situations. In the long term, the browser should be able to properly offload even if mobile network technologies or computing power evolve. We believe that research on offloading is valuable because it has potential to help energy and data usage, not only performance, while we note that the constraint of the three metrics is likely to stay on mobile without external support such as offloading.

In the rest of the position paper, Section 2 identifies the opportunities for offloading and suggests that offloading mobile page loads can help both network and computation.

MCC'13, August 12, 2013, Hong Kong, China.

Section 3 examines the challenges in making offloading decisions and discusses issues raised by using offloading (*e.g.*, broken HTTPS end-to-end security and broken cache). We review the background of the page load process in Section 4 and propose a measurement-based framework that allows to offload any portions of the page load process to the cloud in Section 5. Section 6 reviews related work and Section 7 concludes.

2. OPPORTUNITIES

We begin with examining the opportunities offered by offloading mobile Web pages. Note that offloading mobile Web applications exhibits different opportunities than offloading mobile native applications. This is because Web applications differ from native applications in two aspects: (i) Web pages are downloaded when being requested and thus network activities and computation are inter-dependent, and (ii) computation of different kinds of objects (*e.g.*, HTML and CSS) incurs different patterns. The result is that offloading techniques that have been used for mobile native applications [5, 3, 4] cannot be directly applied to Web applications.

To make the case that offloading is a viable approach to accelerate the mobile Web, we identify three opportunities offered by offloading. This includes i) reduced round trips, ii) reduced time in computation, and iii) consistency with other mobile constraints.

2.1 Reduced round trips

Unlike offloading native applications that introduce additional round trips between the mobile device and the cloud, offloading Web applications can reduce round trips between the mobile device and the cloud.

To demonstrate this opportunity, we first look at how a mobile device interacts with Web servers when loading a Web page. When a page request to `example.com` (fictitious) is made, the mobile browser performs a DNS lookup, sets up a TCP connection, waits for a response, and downloads the response header and content. Often, the request is redirected to another mobile page, *e.g.*, `m.example.com`. Upon receiving the first byte of the root HTML page, the mobile browser iteratively parses the HTML page and requests embedded objects, possibly from other Web servers, *e.g.*, `foo.com` and `bar.com`, until every Web object is fetched, parsed, and loaded.

Consider when portions of the page load process are offloaded to the cloud, similar to redirecting the traffic to a proxy. The cloud interacts with the Web servers on behalf of the mobile device, but the round trip latencies can be much lower when the cloud server is placed near the Web servers. The cloud can connect to the mobile device via a single TCP connection such as SPDY [13] to batch up small HTTP transactions for loading a page. Note that Web page transmission from the cloud to the mobile device can occur after the root HTML page is fetched and does not have to wait for the entire page to be fetched. The notion of leveraging the cloud has been adopted by Opera mini [11], Amazon Silk browser [12], and recently Chrome beta for Android [2], suggesting the feasibility of leveraging cloud in mobile Web.

2.2 Reduced time in computation

Despite the rapid growth in computing power on mobile phones, the cloud or even a high-end desktop is still more powerful in computing as of today. Therefore, offloading computation to the cloud offers the opportunity to reduce page load time. The argument here is similar to code offload for mobile native applications [5, 3, 4], but offloading techniques differ because computation for page loads has certain patterns (*i.e.*, HTML parsing, JavaScript and CSS evaluation, rendering).

We demonstrate this opportunity by measuring the computation time of loading a page on a mobile phone and on a laptop. We use our recently developed tool WProf [15] that captures the dependencies and the time to load and compute each Web object during a page load. The mobile phone is a Nexus S with 1GHz CPU and 512MB memory and the laptop is a Macbook Pro with 2.66GHz CPU and 8GB memory. We find that the time to evaluate the same piece of JavaScript is less than 20 milliseconds on the laptop but is as large as 100 milliseconds on the mobile phone. The time to evaluate the same set of CSS is 100 milliseconds on laptop compared to over 200 milliseconds on the mobile phone. A recent study also suggests that the gap between smartphone and computer is 5.5 to 23.1 times in JavaScript execution speed [7].

While offloading the logics of the page load process to the cloud suggests exciting benefits, Opera mini and Android Chrome beta only compress the Web pages in the cloud while leaving the full logics of page load processing to the mobile device. We consider offloading partial logics of the page load process to the cloud which offers more opportunities.

2.3 Consistency with other mobile constraints

For techniques to be practical on mobile devices, one has to consider their compatibility with other mobile constraints. There are three key constraints on mobile phones, say performance, power consumption, and 3G/4G data usage. Below, we examine whether offloading to the cloud helps relax each constraint. Performance is a key constraint on mobile devices which incur less computing power and slower access network than desktop counterparts. We have demonstrated that offloading can reduce round trips and computation time which in turn can reduce the total page load time. Thus, we do not elaborate here.

Power consumption. Battery power is a scarce resource on mobile devices that limits the deployment of more computing power and energy-intensive applications. We argue that offloading likely results in less power consumption in both computation and network for a page load. It is straightforward that offloading portions of the computation in page loads incurs less power consumption on the mobile device. Besides consuming less power in computation, offloading can also help the power consumed in network. Because the mobile device uses a single TCP connection instead of multiple connections per Web page, less resources are allocated for network. Because the connection between the mobile device and the cloud is under control, the traffic can be shaped to avoid energy inefficiency problems such as 3G long tail.

3G/4G data usage. 3G/4G network operators charge data-plan subscribers based on the used traffic volume, making the 3G/4G data usage a key metric to consider in designing mobile applications and techniques. As portions of the page load process are offloaded to the cloud, the Web page transmitted to the mobile device will be some form of intermediate representations instead of raw Web pages.

For example, it is possible that a DOM tree decorated by matched CSS is delivered to the mobile device in place of an HTML page and some CSS files. It is unclear whether this will increase or decrease the data usage at this time. But, since the intermediate representation can be compressed before being transmitted, there is an opportunity to reduce the transmitted data to the mobile device.

3. CHALLENGES

We have demonstrated the opportunities offered by offloading to accelerate the mobile Web. To design a mobile browser that sufficiently exploits offloading, we need to address the following challenges.

Understanding the mobile page load process. There have been several techniques that aim to accelerate the mobile Web [6], and thus it is surprising to learn that the mobile page load process is still poorly understood. A key reason is the lack of standardization of the page load process while current Web standards focus on enriching the feature set (*e.g.*, HTML 5, CSS 3). The result is that browsers embrace different implementations and each corresponds to millions of lines of code. The rapid evolvement of the Web and browser ecosystem further compounds this situation. Lack of standardization and documentation makes the page load process hard to understand. However, without understanding the page load process it is hard to decide how to offload.

What and when to offload. To fully exploit the power of offloading, a key question to answer is what and when to offload. Offloading portions of the page load process to the cloud makes use of the computing power of the cloud, resulting in less time in computation. Then, the cloud transmits the processed Web page (*a.k.a.*, intermediate representation) to the mobile device which could increase the transmission time if the processed page is significantly larger than the raw page. Thus, the key to maximize the page load performance is to trade off between the decreased computation time and the potentially increased network time. This tradeoff depends on so many factors, namely the Web page, difference in computing power, network characteristics, and the size of intermediate representations, that make up the question of what and when to offloading hard to answer. The Amazon Silk browser [12] only considers the factor of Web page when considering this tradeoff, and thus does not fully exploit the power of offloading.

How to use the cloud. While the benefits by using the cloud is intuitive, it is unclear how to use the cloud to maximize page load performance. It is hard to determine locations of the cloud servers that interact with Web servers. Recall that placing the cloud server near the Web servers incurs minimal round trip latencies. However, it is hard to know what Web servers the cloud server needs to interact with before the cloud server is chosen. Because of the heavy use of content distribution networks (CDNs), the Web servers that host content for a single Web page are dynamic which further complicates the situation. Addressing this challenge requires clever use of the cloud.

Security issues. With the advocacy of HTTPS to protect user privacy on the Web, the cloud in the middle, however, breaks the end-to-end security chain provided by HTTPS. Android Chrome Beta [2] avoids this problem by only working for HTTP pages. The cost is that it does not help on HTTPS pages; and the number of HTTPS pages is arising. The challenge here is to design a reasonable trust model when the cloud is a necessary piece in the security chain.

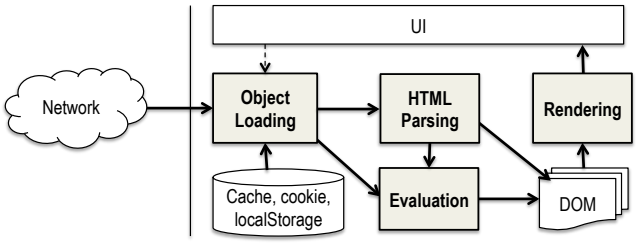

Figure 1: The workflow of a page load. It involves four processes manipulated by different controllers (shown in gray). Object Loader is the only network-facing controller; the other three are computational controllers.

Caching issues. Currently, mobile browsers store Web cache in the form of raw files directly downloaded from Web servers such as HTML, CSS, and JavaScript. However, the cache model is changed when portions of the page load process are moved to the cloud and intermediate representations of Web pages in place of raw pages are transmitted to the mobile device. The problem here is that a piece of intermediate representation (*e.g.*, styled DOM tree) can be the result of multiple raw files (*e.g.*, HTML and CSS) specified with different expiration times. Thus, it is challenging to keep the exact pace of cache expiration by working with intermediate representations on the mobile device.

4. PAGE LOAD PROCESS

Before we describe our proposed offloading approach, we first review the required background of the page load process. Below, we first describe the workflow of a page load and then highlight the page load bottlenecks.

4.1 Workflow

Figure 1 shows the workflow for loading a Web page. The page load starts with a user-initiated request that triggers the *Object Loader* to download the corresponding root HTML page. Upon receiving the first chunk of the root page, the *HTML Parser* starts to iteratively parse the page and download embedded objects within the page, until the page is fully parsed. The embedded objects are *Evaluated* when needed. To visualize the page, the *Rendering Engine* progressively renders the page on the browser. Below, we describe each of these controllers in detail.

HTML Parser: The Parser is key to the page load process, and it transforms the raw HTML page to a document object model (DOM) tree. A DOM tree is an intermediate representation of a Web page; the nodes in the DOM tree represent HTML tags, and each node is associated with a set of attributes. The DOM tree representation provides a common interface for programs such as JavaScript to manipulate the page.

Object Loader: The Loader fetches objects requested by the user or those embedded in an HTML page. The objects are fetched over the Internet using HTTP or SPDY [13], unless the objects are already present in the browser cache. The embedded objects fall under different mime types: HTML (*e.g.*, IFrame), JavaScript, CSS, Image, and Media.

Embedded HTMLs are processed separately and use a different DOM tree. Inlined JavaScript and CSS do not need to be loaded.

Evaluator: Two of the five embedded object types, namely, JavaScript and CSS, require additional evaluation after being fetched. JavaScript is a piece of software that adds dynamic content to Web pages. Evaluating JavaScript involves manipulating the DOM, *e.g.*, adding new nodes, modifying existing nodes, or changing nodes' styles. Since both JavaScript Evaluation and HTML Parsing modify the DOM, HTML parsing is blocked for JavaScript evaluation to avoid conflicts in DOM modification. However, when JavaScript is tagged with an *async* attribute, the JavaScript can be downloaded and evaluated asynchronously in the background without blocking HTML Parsing. The process of JavaScript evaluation generates several intermediate representations in both compilation and execution. For example, the V8 JavaScript engine [14] generates two levels of intermediate representations (*i.e.*, Hydrogen and Lithium) instead of bytecode.

Cascading style sheets (CSS) are used for specifying the presentational attributes (*e.g.*, colors and fonts) of the HTML content and is expressed as a set of rules. Evaluating a CSS rule involves changing styles of DOM nodes. For example, if a CSS rule specifies that certain nodes are to be displayed in blue color, then evaluating the CSS involves identifying the matching nodes in the DOM (*a.k.a.*, CSS selector matching) and adding the style to each matched node. A resulted intermediate representation here is the matched CSS selectors.

Rendering engine: Browsers render Web pages progressively as the HTML Parser builds the DOM tree. Rendering involves two processes–Layout and Painting. Layout converts the DOM tree to an intermediate representation (*i.e.*, the layout tree) that encodes the size and position of each visible DOM node. Painting converts this layout tree to pixels on the screen.

Note that the intermediate representations generated from these controllers are the key to making offloading decisions, because intermediate representations can be larger than the raw Web objects which increases the transmission time.

4.2 Bottlenecks

The four processes in Figure 1 can block each other, resulting in dependencies. We have extracted four kinds of dependencies that are caused by (i) the natural order that activities occur, (ii) the correctness of execution when multiple processes modify a shared resource, (iii) tradeoffs between data downloads and page load latencies, and (iv) limited computation power and network resources [15]. Our previous work suggested that improving only network or computation provides limited benefits to page load time [15]. Unlike many other page load optimization techniques that only help network or computation, offloading shows potential to help both.

The bottlenecks of a page load can be identified by applying critical path analysis to the dependency graph. Identifying bottlenecks are important. This is because improving bottleneck activities can improve the overall page load time but improving non-bottlenecks cannot help page load. The implication here is that the offloading decisions need to consider reducing the time spent on the bottlenecks.

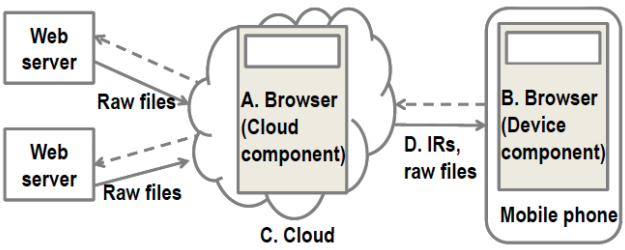

Figure 2: Architecture of the framework.

5. OFFLOADING TO THE CLOUD

This section describes our proposed approach to offload mobile page loads to the cloud. First, we propose the framework that allows to offload portions of the page load process. Note that we propose a framework instead of a design because some design decisions need to be informed by exercising with real-world environments. Second, we plan to use a measurement study on this framework to derive empirical models of offloading strategies. Based on the empirical models, we will develop a mobile browser that selectively offloads portions of page loads by taking real-time measurements of mobile and environments as inputs.

5.1 Framework

We propose a flexible framework that allows to offload portions of the page load process. Figure 2 shows the architecture that consists of three components: a mobile device, a cloud, and Web servers. The browser is split into two browsers, one on mobile (device component) and one in the cloud (cloud component). To fetch a Web page, the mobile browser makes a request to the cloud and the cloud loads the Web page on behalf of the mobile device. Then, the cloud partially processes the page and sends the processed page in the form of processed portions (*a.k.a.*, intermediate representations) and unprocessed portions back to the mobile device via a single TCP connection. Below, we describe the design of key components in this architecture.

5.1.1 Offloading portions of computation

At the core of the framework is offloading portions of the page load process. To be flexible, the framework needs to enable fine-grained division of the page load process. Parts of the divided process are handled by the cloud while the other parts are handled by the mobile device.

Anatomizing the page load process. We start with the anatomy of a page load process. Figure 3 shows the data flow of a page load process that converts raw Web pages to pixels on the screen. The example page contains a CSS, two JavaScripts, and an image in the exact order. The data flow (indicated by gray arrows) is represented by a directed acyclic graph (DAG) that starts with raw files and ends with pixels on the screen. Note that Figure 3 only shows a coarse-grained data flow and does not include all intermediate representations in a page load. A fine-grained data flow can be derived by replacing the boxes with detailed flows.

We equal the problem of dividing the page load process to finding cut(s) of the DAG. One side of the cut is handled by the cloud component and the other is handled by the device component. Note that a cut can mean that a process is partially completed. For example, if there is a cut between the raw CSS file and the DOM styles in Figure 3, it can

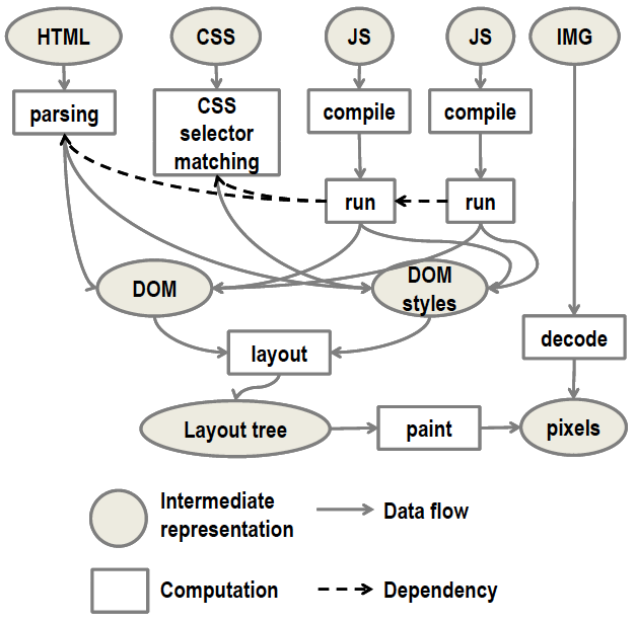

Figure 3: Data flow of processing an example Web page. Note that some IRs for evaluating JavaScript are omitted here.

represent that CSS evaluation has not yet started or that CSS evaluation is almost done.

Constraining the divisions by dependencies. Due to the dependencies within the page load process [15], the cuts are subject to some constraints. For example in Figure 3, the first JavaScript can only be executed after the CSS is evaluated because they both could change the DOM styles and the order is critical here. We indicate the dependencies by dashed arrows. Thus, an admissible cut should satisfy the following constraint: the direction of the dependencies should be in the opposite direction as the data flows through the cut. This constraint ensures a cut to be practical and to represent a real page load division.

Using the division. Although it is possible to divide the page load process into more than two portions and let the cloud and device components handle every other portion, we only consider the case of dividing into two portions. This is because additional divisions increase the interactions between the cloud and the device, and thus increases page load time. We use the cloud to handle the first portion that starts with raw Web objects and we let the device to handle the other portion that ends with pixels on the screen. A possible division would let the cloud compute the DOM and its styles and let the mobile device do the rest. However, if running a piece of JavaScript significantly increases the size of the intermediate representations to transfer, a better division would be running the JavaScript on the mobile device.

5.1.2 How to use the cloud

We identify two key issues in using the cloud: (i) where to place cloud servers and (ii) when to make offloading decisions. We leave the discussions of other issues such as load balancing as future work.

Where to place cloud servers. We have shown that a single TCP connection between the client and the cloud servers is likely to reduce round trip latencies. To this end, we suggest to place the cloud server near or at a CDN service provider. Since the cloud servers in any case need to cache the Web page locally, the latency can be greatly reduced if the servers are located near the CDN service provider. Our proposed framework, a partial browser in the cloud, can act as the front-end of the CDN service which is similar to [8]. In most cases, we expect the requested Web page cached in the cloud so that round trips can be saved. Otherwise, the cloud can promptly detect the rising hot spots and cache those Web pages in the server to reduce the latency in the future.

When to make offloading decisions. The cloud should be in charge of making offloading decisions because it is the first of all active components in our framework that collects all the useful information (*e.g.*, Web pages, and states of the network and the mobile device) to make offloading decisions. We believe that the decisions depend on the diverse situations and Web pages. For instance, for Web pages with complex embedded JavaScripts, offloading the computation to the cloud would be the most effective; if a Web page consists of lots of images and the mobile devices is on a slow network, compressing the image in the cloud would benefit. The problem here is that by the time the first chunk of the raw page is received the cloud could lack the required information to make offloading decisions. Thus, the cloud needs to either wait until it receives more content of Web pages or rely on historical information. The former approach can slow down page load. The latter approach requires us to identify the content of historical information–features that affect offloading decisions. We plan to use experiments to inform whether the former approach is feasible and if not what are the determining factors that affect offloading decisions.

5.1.3 Using a single TCP connection

We use a single TCP connection to transfer partially processed Web page from the cloud to the mobile device. This is because a single TCP connection suggests several benefits over parallel TCP connections. It avoids the slow start phase when opening a new TCP connection, resulting in more bandwidth. While some Web objects (*e.g.*, JavaScripts and HTML) are expected to be loaded with higher priority than some others (*e.g.*, images) [15], parallel TCP connections evenly divide the bandwidth which suggests no priority. Instead, a single TCP connection can multiplex Web objects using specified priority policies.

A hands-on protocol of a single TCP connection that multiplexes Web objects is SPDY [13]. To make the maximum benefits out of a single TCP connection, the cloud servers require some other configurations. For example, while the TCP initial window size is set to 10 as of Linux kernel 2.6.39 suggested by RFC3390 (and it was 3), configuring a larger TCP initial window can avoid more slow start. The specified priority policy needs to consider the dependencies shown in Figure 3. Intuitively, intermediate representations or raw pages that are depended by more resources should result in higher priority.

5.1.4 Others

To address the HTTPS issue, we can establish two TLS/SSL connections, one between the Web server and the cloud and the other between the cloud and the mobile device. This enlarges the trust base by including the cloud,

providing less strong security than end-to-end HTTPS. But, to process the Web pages the cloud needs to know some form of the page content which in turn weakens security. We leave the caching issue to the future.

5.2 Measurements

To systematically determine the optimal strategies for given situations, we plan to conduct extensive measurements with this framework. We will consider all the situations that contribute to the tradeoffs between faster computation and potentially slower network transmission offered by offloading. As for network, we consider varying RTTs, bandwidths, and packet loss rates to the cloud that are imposed by choices of WiFi, 3G, or 4G and by different vendors. The availability of cloud servers near the Web servers is also a key factor. As for computation, we consider the differences in computing power (*e.g.*, CPU speed and memory) of the mobile device and of the cloud.

Based on the measurement results, we will derive empirical models of offloading strategies that take measurements of the mobile and cloud environments as inputs. Because the empirical models will only depend on measurements not the technologies themselves, our approach is able to accommodate situations even if mobile network technologies or computing power evolve. The outputs of empirical models will be cuts of data flow in Figure 3 that yield minimal page load times.

5.3 Cloud-based mobile browser

Informed by the empirical models, we will develop a browser that selectively offloads portions of the page load process depending on real-time measurements of the situations. In this way, mobile page loads can benefit from offloading at any situation, or browsers can turn off offloading if it does not benefit page loads. For example, the browser had better offload JavaScript evaluation followed by a user-initiated click if the script will take a long time to run on mobile. Unlike Opera Mini and Android Chrome Beta, our browser will offload partial page load process; and unlike Amazon Silk, our browser will consider ambient situations when making offloading decisions.

6. RELATED WORK

A large body of previous work has proposed to offload parts of mobile native applications from mobile devices to the cloud for the benefits of performance [5, 3] and energy [4]. Gordon et al. [5] introduced a runtime system to migrate unmodified Android applications freely between machines. Chun et al. [3] allowed part of an application to be cloned and executed in a computational cloud. Cuervo et al. [4] offloaded smartphone code to save energy. While they focus on offloading native applications, their proposed techniques are specific to platforms and thus are not applicable to the Web.

Surprisingly, little has been done to offload mobile Web applications in the research community. The industry has pioneered in this field with notable examples of the Opera Mini browser [11], Android Chrome Beta [2], and the Amazon Silk browser [12]. Opera Mini compresses pages to a markup format called OBML that reduces both transfer time and data usage. Android Chrome Beta applies techniques such as mod_pagespeed [9] to the pages. But, they both still leave the full logics of page load processing to mobile devices. The Amazon Silk browser moves portions of page loads to the cloud which depend on the Web page itself. However, it is unclear how the Silk browser divides the page load process and the choice of offloaded portions should be a result of ambient situations, not only Web pages. We advocate an elastic framework that allows to offload any portions of the page load process; the offloaded portions are subject to ambient situations such as network speed, not only Web pages.

7. CONCLUSION

This position paper proposes a measurement-based framework that allows to offload any portions of mobile page load process to the cloud. We will experiment with a large variety of real-world situations by offloading varying portions of page loads using our framework. Informed by the experimental results, we will develop a mobile browser that considers the diverse situations as well as energy and data usage.

8. REFERENCES

[1] Alexa. `http://www.alexa.com/`.

[2] Data compression in chrome beta for android. `http://blog.chromium.org/2013/03/data-compression-in-chrome-beta-for.html`.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of Eurosys, 2011*.

[4] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code ofSSoad. In *Proceedings of Mobisys, 2010*.

[5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of OSDI, 2012*.

[6] S. Hadjiefthymiades and L. Merakos. Using proxy cache relocation to accelerate web browsing in wireless/mobile communications. In *Proceedings of WWW, 2001*, New York, NY, USA.

[7] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE NEtworks. In *Proceedings of Mobisys, 2012*.

[8] F. T. Leighton and D. M. Lewin. Html delivery from edge-of-network servers in a content delivery network (cdn). In *US Patent 7996533, August 2011*.

[9] mod_pagespeed. `http://www.modpagespeed.com/`.

[10] Opera binary markup language. `http://dev.opera.com/articles/view/opera-binary-markup-language/`.

[11] Opera mini browser. `http://www.opera.com/mobile/`.

[12] Amazon silk browser. `http://amazonsilk.wordpress.com/`.

[13] Spdy. `http://dev.chromium.org/spdy/spdy-whitepaper`.

[14] V8 javascript engine. `https://code.google.com/p/v8/`.

[15] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Proceedings of NSDI, 2013*.