Why are Web Browsers Slow on Smartphones?

[1]Zhen Wang, [2]Felix Xiaozhu Lin, [1,2]Lin Zhong, and [3]Mansoor Chishtie
[1]Dept. of ECE and [2]Dept of CS, Rice University, Houston, TX 77005 [3]Texas Instruments, Dallas, TX

ABSTRACT

We report the first work that examines the internals of web browsers on smartphones, using the WebKit codebase, two generations of Android smartphones, and webpages visited by 25 smartphone users over three months. We make many surprising findings. First, over half of the webpages visited by smartphone users are not optimized for mobile devices. This highlights the importance of client-based optimization and the limitation of prior work that only studies mobile webpages. Second, while prior work suggests that several compute-intensive operations should be the focus of optimization, our measurement and analysis show that their improvement will only lead to marginal performance gain with existing webpages. Furthermore, we find that resource loading, ignored by all except one prior work, contributes most to the browser delay. While our results agree with a recent network study showing that network round-trip time is a major problem, we further demonstrate how the internals of the browser and operating system contribute to the browser delay and therefore reveal new opportunities for optimization.

1. Introduction

As one of the most important applications on smartphones, the web browser is known to be slow and often take seconds or tens of seconds to open a page. Understanding why the browser is slow is critical to its optimization. We are motivated by two recent research endeavors. First, many have studied browsers on personal computers and concluded that several key compute-intensive operations are the bottleneck [2-4]. On the other hand, a recent smartphone characterization study [1] demonstrated that the wireless hop can significantly slow down the browser by its long round-trip time (RTT). However, the authors took a black-box approach without looking into the internals of the web browser, thereby provided limited insights.

In this work, we examine the internals of web browsers on smartphones with two novel methods. *(i)* We analyze webpages visited by 25 iPhone 3GS users over three months. The analysis reveals that over half of the webpages visited are not optimized for mobile devices. Therefore, although mobile webpages do make browsers faster, they are only half of the story. *(ii)* We are the first to utilize two techniques to analyze the browser performance, *dependency timeline characterization* and *what-if analysis*, based on instrumenting the popular WebKit source code [5]. We are able to truthfully capture the user-perceived delay of opening a webpage, reveal the dependency and concurrency of browser operations, and evaluate the impact of possible optimizations, which are impossible using techniques employed in prior work.

HotMobile'11, March 1–2, 2011, Phoenix, Arizona, USA.

We make the following key findings. (*i*) Improvement on compute-intensive operations suggested by prior work such as *style formatting*, *layout calculation* [2-4], and *JavaScript execution* [1] will lead to marginal improvement in browser performance on smartphones. (*ii*) Instead, *resource loading* is the key to browser performance on smartphones. Resource loading is the process that resources, usually files of various types needed by opening a webpage, are acquired by the smartphone from the web server. In contrast, prior work [2-4] assume resource loading contributes negligible delay, which is not true with smartphones. (*iii*) Given a resource, the delay of resource loading is determined by the network condition, the browser loading procedure and the processing power of the smartphone. Our results agree with the findings from [1] that long network RTT is detrimental to the browser performance. We further find that improvement in network bandwidth will not improve browser performance much beyond typical 3G network. Finally, by comparing the behaviors of two smartphones, Google Nexus One (N1) and HTC Dream (G1), we observe a more powerful hardware, e.g. N1, will reduce the browser delay mainly by accelerating OS services and network stack, instead of the compute-intensive operations suggested by prior work.

Our findings not only shed light into the behavior of web browsers on smartphones but also have important implications to optimization. In particular, our work suggests that one should aggressively seek to hide the network RTT and improve OS services and network stack in order to improve resource loading, instead of optimizing the compute-intensive browsers operations, such as layout calculation, style formatting, and scripting suggested by prior work [2-4].

The rest of the paper is organized as follows. Sections 2 and 3 provide the background for the web browser operations and discuss related work, respectively. Section 4 presents our characterization methodology, i.e., dependency timeline characterization and what-if analysis. Section 5 describes our experimental setup and Section 6 offers important findings. Section 7 discusses the implications of our findings to optimizing web browsers on smartphones.

2. Background

A modern browser is a very complicated piece of software. For example, the WebKit source code in Android 2.1 has around one million lines in over 5,700 files [5]. We next provide an architectural overview of WebKit-based browsers.

When opening a page, the browser incrementally loads multiple web resources, builds an Internal Representation (IR) of multiple loaded resources, and converts the IR to the graphical representation. A web *resource* is an individual unit of content or code such as HTML documents, Cascading Style Sheets (CSS), pictures, and JavaScrip files. Typically, an IR employs a set of tree structures to record different information of hierarchical Document Object Model (DOM) elements, which correspond to the various HTML elements in the webpage such as paragraphs, images, and form fields.

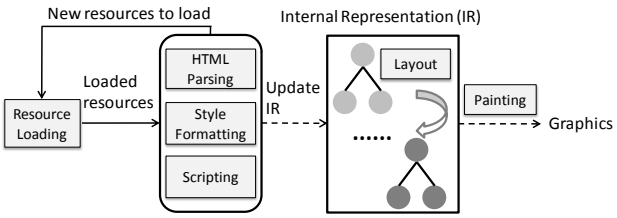

Figure 1: The procedure of opening a webpage

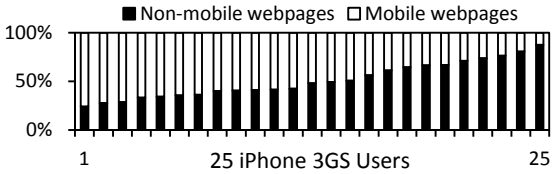

Figure 2: The percentages of mobile and non-mobile web page visited by each of the 25 iPhone 3GS users

The procedure of opening a page (illustrated by Figure 1) involves a set of interdependent operations that can be dynamically scheduled and concurrently executed. The operations can be classified into three categories. The first category includes *resource loading*, which fetches a resource given its URL, either from the remote web server or local cache. Resource loading uses services from the underlying network stack, e.g., resolving the domain name in the URL, handling HTTP URL redirection, establishing TCP connections and so on. Given the resource, the latency of resource loading is determined by the network RTT, network bandwidth, browser loading procedure and the processing power. The second category includes five IR operations that produce the IR by processing loaded resources and consume the IR to render the page. The five operations are *HTMLParsing* (or *Parsing*), *StyleFormatting* (or *Style*), *Scripting*, *Layout*, and *Painting*. The first three process HTML documents, style constraints (e.g., CSS), and JavaScript, respectively, and attach results incrementally to the IR. *Layout* computes and updates the screen locations of DOM elements based on the recently updated IR. *Painting* employs the IR to generate the final graphical representation of the web page. Finally, all other processing incurred by the browser is treated as one operation, called *Glue* operation, in this paper.

While it is tempting to think the first six operations described above as a pipeline, three key properties of them make the page opening procedure far more complicated than a pipeline. First, when opening one webpage, an operation can be performed many times. For example, it often takes *multiple* loading-processing iterations over *multiple* resources to finish opening a web page. This is because the browser discovers new resources while processing loaded ones. Second, operations can be *concurrently* executed. For example, there can be multiple instances of *resource loading* on-going at the same time; in the meantime, they may overlap with other operations such as *Scripting* and *Layout*. Third, operations are dynamically scheduled. For example, with several recent updates to the IR, the browser determines when to trigger a *Layout*; the completion of loading a resource leads to its processing, and the browser determines when to process; *Parsing* encounters new URLs in a document, and the browser decides whether to request them immediately or not.

2.1 Mobile Webs

Many web sites provide a mobile version of their pages in order to better fit the content into and better support navigation on the small screen [6]. In general, the mobile version of a webpage is not as content-rich as the original one intended for personal computers. Mobile webpages tend to have smaller CSS files and fewer JavaScripts that lead to lighter workload for Style and Scripting.

While webpages optimized for mobile devices are usually faster to open, we find that a significant portion of webpages visited by smartphone users are not optimized for mobile devices, either because the mobile versions are unavailable or the users purposefully choose the non-mobile versions for their richer content. Using the traces collected from 25 iPhone 3GS users over three months [7], we found that over half (56%) of the pages visited were not optimized for mobile devices, or *non-mobile webpages*, as illustrated in Figure 2. Therefore, our characterization will consider both mobile and non-mobile web pages. In contrast, the only prior work characterizing smartphone browsers mostly uses mobile webpages [1].

The fact that over half of the webpages visited by smartphone users are not optimized for mobile devices further highlights the importance of client-based optimization, because many websites will simply not provide an optimized mobile version and users often prefer information-rich non-mobile pages.

3. Related Work

The performance of browsers has attracted quite a lot of interests from both industry and academia. Existing work, however, is limited in both its scope and methods. Characterization work on PC browsers assumes resource loading is negligible for PCs with enterprise Ethernet and therefore focuses on the compute-intensive IR operations [2-4, 8] . Internet Explorer (IE) team [2] provided a breakdown of the CPU cycles consumed by the key IE subsystem, which focused on the computation of the browser. The network improvement is discussed separately and is not clearly included in the breakdown. Using call stack sampling for performance characterization, the authors of [3, 8] threw out the network time in their analysis since their profiling method cannot capture the time spent idling. None of their methods capture the cost of resource loading or consider the concurrency of operation execution as discussed in Section 2. In contrast, we will show that even when enterprise Ethernet is used, optimizing the IR operations will only lead to marginal overall improvement because resource loading contributes most to the critical path in the browser delay on smartphones.

Huang et al. [1] investigated smartphone browser performance mainly from the network perspective without looking into the internals of the browser. They quantified how the browser performance is affected by network RTT, packet loss rate, concurrent TCP connection, and resource content compression, without an understanding of how the browser operates and interacts with smartphone OS and network. The authors, however, measured the browser performance with "the time between the first DNS packet and the last data packet containing the payload from the server". This measurement will not accurately capture the user-perceived latency. First, it missed the latency of the browser initialization for the page before the first DNS packet out (typically around 200ms on G1) and all operation executions after the reception of the last packet, which can take up to 2 seconds according to our measurement. Moreover, the authors approximated computing time by the browser with the TCP idle time, or periods having no network activity. This approximation will miss a significantly portion of computing time by the browser (up to 40% according to our observation) because many IR operation instances are executed in parallel with network activities. We find that an accurate understanding

of the browser delay absolutely requires an examination of the browser internals.

4. Characterization Methodology

To capture the user-perceived browser performance, we calculate the *browser delay* as follows: the starting point is when the user hits the "GO" button of the browser to open an URL. The end point is when the browser completely presents the requested webpage to the user, i.e., the browser's page loading progress bar indicates 100%. Such latency covers the time spent in all operations involved in opening a page, and can be unambiguously measured by keeping time-stamps in the browser code. We note that modern browsers utilize incremental rendering to display partially downloaded webpage to users. We do not consider partially displayed webpage as the metric because it is subjective how partial is enough to say the webpage is opened.

Two questions are critical for in-depth understanding of the browser delay and potential optimizations: 1) how do various operations collectively contribute to the browser delay; 2) what is the overall performance improvement if certain operations are accelerated. To answer these two questions, we employ two methods, called dependency timeline characterization and what-if analysis, described below.

4.1 Dependency Timeline Characterization

The *dependency timeline graph* for opening a webpage is a two-dimensional diagram that visualizes all *operation instances*, as shown in Figure 3. The dependency timeline graph reflects the temporal relations by arranging all operation instances along the X axis (i.e., the time axis). Furthermore, it organizes operation instances into *resource groups* along the Y axis. In each resource group, the operation instances either load or process a common resource. Instances of Layout and Painting operations have their individual groups because they are not directly related to any resource. For example, *Painting* only consumes the most updated IR. Resource groups reveal important dependencies and concurrencies among operation instances. Within a group, an instance directly depends on its predecessor, as they have to be executed sequentially. Additionally, a group, G, is dependent on an operation instance from another group if G's resource is discovered by that instance.

The dependency timeline graph visualizes both intra-group and inter-group dependencies: intra-group dependencies are shown along the same horizontal level; inter-group dependencies are indicated by dashed lines. The graph provides two key insights into the browser performance. First, it offers the detailed latency breakdown at the operation-level, by including timestamps of important functions in all operation instances. Second, the dependencies serve as the foundation of the what-if analysis (Section 4.2). To the best of our knowledge, we are the first to visualize such dependencies using real traces.

To capture the dependency timeline, we added about 1200 lines of code to 27 files of WebKit. For important functions in each operation, we log information including timestamp, function name, resource name, etc. For example, for each resource loading instance, we log such information when the loading request is scheduled, sent out, the response is received, and the resource is loaded. All logs are kept as compact data structures in memory and only saved to the non-volatile storage after the page opening ends in order not to add any file I/O latency. After the experiment, we parse the log to construct the dependency timeline. We have verified that the instrumentation code contributes negligible latency (<1%) to the browser delay.

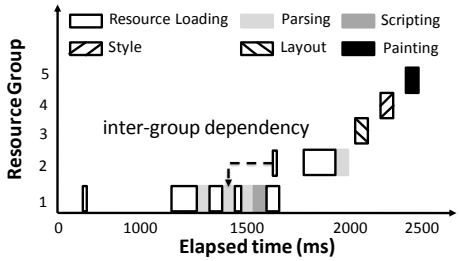

Figure 3: A simplified dependency timeline graph

The proposed dependency timeline is motivated by the timeline panel [9] provided by WebKit [5]. However, the timeline panel cannot provide the complete dependency relationship among different operation instances. Furthermore, it only works for desktop Safari and Chrome at the time of this writing.

4.2 What-if Analysis

The dependency timeline provides a solid foundation for us to answer an important question: *what* overall performance gain will be achieved *if* a browser operation is accelerated? Our technique is therefore called what-if analysis, which works as follows. To accurately predict the impact of accelerating all instances of any operation, we scale the execution time of each instance of such an operation in the dependency timeline, and shift all its dependant operation instances to the left of the time axis (i.e. executed earlier). The dependency information provided by the dependency timeline determines how much an instance can be shifted. There are three cases:

- If the shifted instance is not the beginning of a resource group, it can shift the same amount of time as its predecessor.
- If the shifted instance is the beginning of a resource group and the group's resource is discovered by another instance, the shifted instance can shift the same amount of time as the instance that discovered the resource.
- If the shifted instance is an IR-consuming operation (Layout or Paint), it will shift the same distance as the most recent IR-producing operation instance does.

5. Experimental Setup

We next describe the experimental settings.

Smartphone Platforms: We study two smartphones, Google Nexus One (*N1*) and HTC Dream (*G1*). We choose these two smartphones in order to see the impact of hardware because they have largely identical software configurations and are from the same original equipment manufacturer (OEM). N1 has a 1GHz Qualcomm Snapdragon Application Processor while G1 has a 528MHz Qualcomm MSM7201A Application Processor. Both smartphones run identical software stacks: the Android 2.1 operating system with our instrumented WebKit.

Network Conditions: We measure the browser delay under three types of networks: emulated enterprise Ethernet, typical 3G network, and emulated adverse network. To emulate *enterprise Ethernet* and *adverse network*, we reversely tether the smartphone through a dedicated gateway, an Ubuntu Linux laptop. The smartphone is connected to the gateway through USB; the gateway is connected to the 1Gbps Rice campus network. With this setup, all the network traffic of smartphone web browsing is forwarded by the gateway. Our measurement shows that the gateway itself has

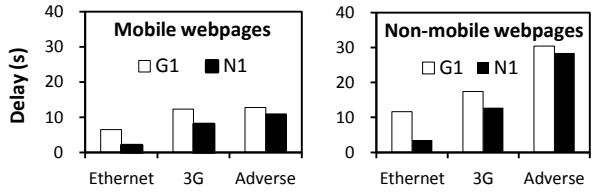

Figure 4: Average browser delay for opening mobile and non-mobile webpages on G1 and N1 through three different networks. Adverse network is emulated with 400ms injected delay in RTT and 500Kbps/100Kbps downlink/uplink bandwidth [1]

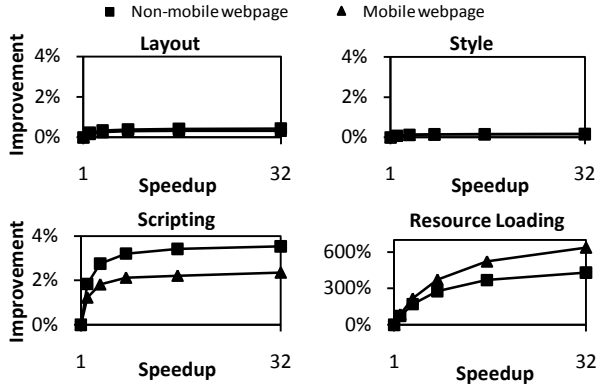

Figure 5: Browser performance improvement when speeding up various operations using 3G (N1)

negligible impact on the network performance: the average RTT between the smartphone and the gateway's Ethernet interface is 1ms; the forwarding bandwidth provided by the gateway is 54Mbps, both of which are too good to be the limiting factors of the end-to-end network performance. The average RTT from the smartphone to top 10 mobile websites [10] is 23ms. To examine the impact of network RTT and throughput and emulate adverse networks, we control the gateway to add extra latency to the end-to-end RTT and throttle the network bandwidth, using Linux Traffic Control. To measure the browser performance with typical 3G networks, we use the 3G network service provided by T-mobile. In order to have a relatively consistent network condition, we always perform the measurements during the midnight and at the same location that sees a strong signal. The average RTT from the smartphone to top 10 mobile websites [10] is 276ms for the 3G network as we measure.

Benchmark Webpages: We employ two sets of benchmark webpages. The *mobile* set includes the mobile versions of the 10 most visited websites from mobile phones as reported in [10]. The *non-mobile* set consists of the 10 most visited non-mobile webpages from 25 iPhone 3GS users in three months, collected from an ongoing field study reported in LiveLab [7]. These webpages were visited 2611 times during the three months.

PageCycler: We implement a smartphone tool called *PageCycler* to invoke the smartphone browser to visit the URLs in a given set one by one. PageCycler also utilizes tcpdump [11] on the smartphone to record the network traffic, e.g., TCP packets, when opening a page. According to our measurements, the overhead of tcpdump is negligible (<2% of CPU time and <0.4% of memory).

6. Characterization Results

We next present findings from the characterization study. Not surprisingly, mobile browsers are slow, even for mobile web pages. Figure 4 presents the average browser delay on N1 and G1 under three different network conditions for two benchmarks.

(*i*) Mobile browsers are slow, especially for non-mobile webpages. Even with Ethernet, the average browser delay to open the non-mobile webpages on N1 is close to four seconds, far from that required for a smooth user experience.

(*ii*) The browser delay is significantly shorter (~30%) on N1 than G1, indicating that more powerful hardware does help. Yet how the hardware helps the performance is not as obvious as it may seem to be.

In the rest of this section, we seek to answer three important questions: 1) What contribute to the browser delay? 2) Where can significant improvement come from? and 3) How does the hardware difference between N1 and G1 make a difference in the browser delay? In order to answer the above questions, we next employ what-if analysis described in Section 4.2 to evaluate the impact of accelerating browser operations in various ways. Our results highlight the limitations of prior work on browser performance characterization.

6.1 IR Operations Do Not Matter Much

Prior work on browser performance characterization and optimization has suggested that optimizing some of the IR operations would be profitable because they are compute-intensive. For example, the IE8 team [2] focused on Layout and Painting; the authors of [3] focused on Layout; and the authors of [4] focused on Layout and Style. Their conclusions are based on counting the CPU usage by these operations, instead of how these operations contribute to the overall performance. In contrast, our what-if analysis reveals that improving these IR operations will only lead to marginal browser delay improvement.

Figure 5 shows the what-if analysis of the mobile browser performance for N1 under typical 3G networks. The X-axis is the speedup applies to the operation. The Y-axis is percentage improvement in the browser performance. We can clearly see that even with 32x speedup of Layout, Style, and Scripting, the browser performance will be improved by 0.4%, 0.2%, and 4%, respectively. Note that 32x speedup would require significantly advancement in hardware and algorithm. For example, the new JavaScript engine V8 [12] introduced in Android 2.2 can improve scripting by only 2-3x [13], resulting in 3% browser performance improvement according our analysis. This result also indicates that the JavaScript benchmark used by [1] may not be representative of JavaScript encountered by smartphone browsers in the field. With Ethernet, the overall improvement from speeding up computing is higher than that with 3G as expected. Yet the conclusions drawn using typical 3G networks still hold. With 32x speedup of Layout, Style and Scripting, the browser performance will be improved by 2.0%, 1.4% and 8.4%, respectively.

One may ask: *will the profit only materialize when all the IR operations are accelerated due to concurrency?* The answer is No. 32x speedup of all five IR operations improves the browser performance by 7% with 3G and by 23% with Ethernet. 32x speedup of all five IR operations plus the glue operation improves the browser performance by 8% with 3G and by 27% with Ethernet.

6.2 Resource Loading Rules

What-if analysis reported in Figure 5 demonstrates that the source of the browser performance problem is in resource loading: 2x speedup of resource loading will improve the browser perfor-

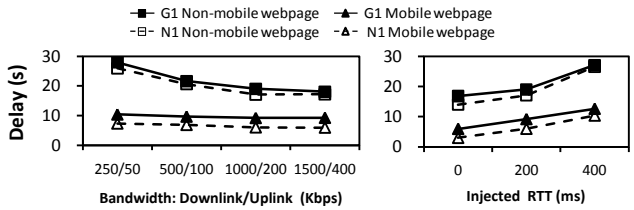

Figure 6: Impact of bandwidth limitation and network RTT on browser delay for N1 (dashed lines) and G1 (solid lines)

mance by over 70%. Due to the importance of resource loading, we zoom into it with a series of measurements to understand it better. As described in Section 2, resource loading fetches a resource given its URL, either from the remote web server or from the local cache. In this process, resource loading uses services from the underlying network stack, e.g., resolving the domain name of the URL, handling HTTP URL redirection, establishing TCP connections and so on. The first resource to load is usually the HTML document. Once a resource is loaded and parsed/scripted, the browser may discover new resources and will schedule the requests to them according to their priorities (determined by the resource file type). Not surprisingly, opening a webpage may require loading multiple resources in series. This explains why resource loading is really important to the browser delay.

Given the resources, the latency contribution from resource loading is determined by four factors: the network RTT, the network bandwidth, resource loading procedure, and processing power available at the smartphone. We next examine how these four factors impact the overall browser delay.

6.2.1 Network RTT and Bandwidth

Using the Ethernet setup described in Section 5, we inject various RTT delays and set various bandwidth limits to emulate the impact by network RTT and bandwidth. While varying one of the two metrics, we fix the other to the typical value in the 3G network [1]: a RTT of 200ms, and a bandwidth of 1000Kbps downlink and 200Kbps uplink. For RTT, we inject RTT from 0ms to 400ms. For downlink/uplink bandwidth, we set the limits from 250/50, 500/100, 1000/200 to 1500/400, all in Kbps. Figure 6 presents the measurement results.

We make the following observations.

(i) Improving the bandwidth does not improve the browser delay much after 1000/200Kbps for downlink/uplink, as shown in Figure 6. Therefore, current typical 3G networks can be considered as adequate since their throughputs are usually in this level or much higher [1]. This is not surprising because the size of all resources for a webpage is usually a few hundred KB (160KB and 770KB for mobile and non-mobile webpages, respectively), according to our measurements. However, as webpages are likely to become richer and therefore come with larger resource files in the future, bandwidth improvement will certainly help.

(*ii*) Network RTT is a key factor to the browser delay as also observed by the authors of [1]. As shown in Figure 6, the browser delay increases significantly when injected RTT increases from 0ms (Ethernet) to 200ms (typical 3G) to 400ms (adverse 3G). The findings regarding the impacts by bandwidth and RTT imply that the browser delay difference between Ethernet and typical 3G, as shown in Figure 4, should be attributed to the difference in RTT rather than that in bandwidth.

6.2.2 Resource Loading Procedure

Resource loading procedure is how the browser loads the resources needed when opening a webpage. Opening a webpage incurs loading multiple resources. On average, there are 21.8 resources for mobile benchmark websites and 96.4 resources for non-mobile benchmark webpages. They are not fully parallelized due to the following loading procedure factors: (*i*) New resources can only be discovered while parsing a loaded resource, e.g., the main HTML file. (*ii*) Redirections on the main HTML file further delay the discovering time of later resources. (*iii*) If there are JavaScripts used, the parsing of the HTML file will be blocked until the JavaScripts have been executed. A side parser [14] can help in this situation, but it is not yet widely used. (*iv*) Finally, the limited number of concurrent TCP connections and sequential secure connection (HTTPS) establishment further serialize the loading of multiple resources.

The loading of each resource incurs multiple network round trips in series, due to redirection, DNS query, TCP connection establishment, creating secure connection (HTTPS) and downloading the resource file. Typically, each resource incurs 3 round trips for HTTP file and 5 round trips for HTTPS. The actual number of round trips varies according to the real situations, such as redirection, TCP slow start, domain name already resolved and TCP connection reuse. Under current resource loading procedure, on average 18.6 round trips are incurred in series for opening top 10 mobile websites and 27.2 round trips are incurred in series for top10 non-mobile webpages. Such large numbers of round trips in series are the key reason that the network RTT matters very much to the browser delay, which was discussed in the previous section.

6.2.3 Processing Power

The resource loading time also depends on the processing power available at the smartphone since it involves network stack and OS service on the smartphone. A careful examination of the dependency timeline graphs reveals the three time intervals in resource loading in which N1 significantly outperform G1. Figure 7 illustrates these three time intervals for mail.yahoo.com.

(i) Time between a SendResourceRequest made by WebKit and when the resource's corresponding request packet is sent out. The request packet can be a query packet for DNS lookup to resolve the domain name, a TCP SYN packet to establish the connection, or an HTTP GET packet to request the resource. During this time, necessary OS services and the network stack are invoked. The time is incurred at the beginning of the loading of each resource.

(ii) Time between when TCP connection for one resource is established and when the HTTP GET is sent out. During this time, the OS notifies the browser when the connection is up, and the browser invokes the network stack to send out the HTTP request. This time is incurred for each round trip.

(iii) Time spent to send a series of back-to-back requests for resource 2-5. During this time, the browser retrieves buffered requests and sends them out by invoking necessary OS services (e.g. resolve the domain name, establish TCP connection, send packet, etc). This time is incurred when multiple resources are requested at the same time.

Apparently the time intervals of *(ii)* are much more frequent but shorter than those of *(i)*. Figure 8 presents the time G1 and N1 spent in opening mail.yahoo.com. N1 is much faster than G1. The loading time reduction is further amplified through multiple serial instances of resource loading when opening a webpage. The total

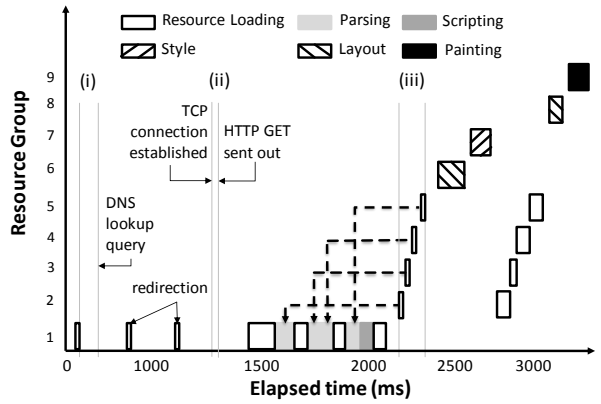

Figure 7: Dependency timeline graph for opening mail.yahoo.com with Ethernet network condition on G1. (i) is incurred for each resource; (ii) is incurred for each network round trip; (iii) is incurred when multiple resources are requested at the same time.

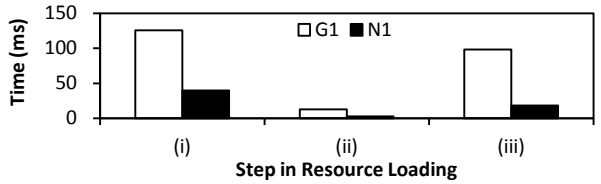

Figure 8: Time spent by G1 and N1 for three steps in resource loading

time spent in the three intervals will be 1.1 seconds on average for N1 and 2 seconds for G1 when opening a mobile webpage.

Based on the findings above, we conclude that more powerful hardware improves the browser delay mainly through faster OS services and network stack instead of faster browser IR operations.

7. Discussions and Conclusions

Our results show that lessons from the *wired* Web [2-4] are not broadly applicable. In particular, our characterization study suggests the most effective way to improve the browser delay for the *wireless* Web is to either reduce resource loading time, in particular the network RTT, or hide its impact. To reduce network RTT, cloudlet [15] and data staging [16] can be employed to move website contents to nearby servers.

There are a few ways to hide the impact of resource loading. *Speculative resource loading* can parallelize the loading of multiple resources and reduce the impact from the network RTT. This is, however, a very challenging problem; since all needed resources have to be preloaded to make the speculative loading effective. Any un-preloaded resource will be in the critical path and largely impact the overall performance. Making one mistake or missing a single resource in the preloading procedure could significantly impact the overall improvement. Speculative resource loading is different from web prefetching [17], which downloads webpages likely to be accessed by the user in the future. On the other hand, as multicore processors are appearing on smartphones, their parallelism can be exploited for speculative resource loading. One core can be dedicated to "coarse-grain" parsing, which aims at getting the URLs of later resources instead of generating the DOM elements. This process should be light-weight and very fast. Thus, the resource requests can be sent out as early as possible.

The loading of multiple resources can be batched in various ways to hide the resource loading time. For example, Google normally batches multiple pictures into one and send it to the browser directly. This eliminates multiple RTTs needed to get several pictures. Furthermore, the batched loading can be supported by a proxy in the cloud in order to suppress the long RTT of the wireless first hop [18]. Finally, it can be supported by new ways of specifying webpage resources so that the browser can load resources as soon as possible, such as Data URI scheme [19].

Acknowledgements

This work is supported in part by the Texas Instruments Leadership University program and by NSF Awards #0803556 and #0923479. The authors are grateful to the anonymous reviewers and the paper shepherd, Dr. Bill Schilit, for their suggestions that helped improve the final version.

References

[1] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)* San Francisco, California, USA: ACM, 2010.

[2] C. Stockwell, "IE8 Performance," http://blogs.msdn.com/b/ie/archive/2008/08/26/ie8-performance.aspx, 2008.

[3] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proc. Int. Conf. World Wide Web (WWW)* Raleigh, North Carolina, USA: ACM, 2010.

[4] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proc. Int. Conf. World Wide Web (WWW)* Raleigh, North Carolina, USA: ACM, 2010.

[5] WebKit, "The WebKit Open Source Project," http://webkit.org/.

[6] S. Shrestha, "Mobile web browsing: usability study," in *Proc. Int. Conf. Mobile Technology, Applications, and Systems and Int. Symp. Computer Human Interaction in Mobile Technology* Singapore: ACM, 2007.

[7] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: Measuring Wireless Networks and Smartphone Users in the Field," in *Proc. Workshop on Hot Topics in Measurement & Modeling of Computer Systems*, June 2010.

[8] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik, "Parallelizing the Web Browser," in *1st USENIX Workshop on Hot Topics in Parallelism*, 2009.

[9] P. Feldman, "WebKit Timeline Panel," http://webkit.org/blog/1091/more-web-inspector-updates/#timeline_panel.

[10] Nielsen.com, "Top mobiel phones, sites and brands for 2009," http://blog.nielsen.com/nielsenwire/online_mobile/top-mobile-phones-sites-and-brands-for-2009/, 2009.

[11] TCPDUMP: http://www.tcpdump.org/.

[12] Google, "V8 JavaScript Engine," http://code.google.com/p/v8/.

[13] Google Android, "Android 2.2 Platform Highlights," http://developer.android.com/sdk/android-2.2-highlights.html.

[14] A. Koivisto, "Optimizing Page Loading in the Web Browser," http://webkit.org/blog/166/optimizing-page-loading-in-web-browser/.

[15] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, pp. 14-23, 2009.

[16] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan, "Data Staging on Untrusted Surrogates," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* San Francisco, CA: USENIX Association, 2003.

[17] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve World Wide Web latency," *SIGCOMM Comput. Commun. Rev.*, vol. 26, pp. 22-36, 1996.

[18] Skyfire: http://www.skyfire.com/.

[19] L. Masinter, "The "data" URL scheme," http://tools.ietf.org/html/rfc2397, 1998.