RICE UNIVERSITY

Speeding Up Mobile Browsers without Infrastructure Support

by

Zhen Wang

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE

Lin Zhong, Chair
Associate Professor
Electrical and Computer Engineering

Dan Wallach
Associate Professor
Computer Science

T. S. Eugene Ng
Associate Professor
Computer Science

HOUSTON, TEXAS
APRIL 2012

ABSTRACT

Speeding Up Mobile Browsers without Infrastructure Support

by

Zhen Wang

Mobile browsers are known to be slow. We characterize the performance of mobile browsers and find out that resource loading is the bottleneck. Leveraging an unprecedented set of web usage data collected from 24 iPhone users continuously over one year, we examine the three fundamental, orthogonal approaches to improve resource loading without infrastructure support: caching, prefetching, and speculative loading, which is first proposed and studied in this work. Speculative loading predicts and speculatively loads the subresources needed to open a webpage once its URL is given. We show that while caching and prefetching are highly limited for mobile browsing, speculative loading can be significantly more effective. Empirically, we show that client-only solutions can improve the browser speed by 1.4 seconds on average. We also report the design, realization, and evaluation of speculative loading in a WebKit-based browser called Tempo. On average, Tempo can reduce browser delay by 1 second (~20%).

.

Acknowledgements

I would like to thank my advisor, Professor Lin Zhong, for his guidance and encouragement during my study and research at Rice University. He has not only given me insightful suggestions, but also helped me to develop the right way to do research.

I am also grateful to work with Mansoor Chishtie from Texas Instruments, who supports my research and gives me inspiring advice.

I would like to thank Professor Dan Wallach and Professor T. S. Eugene Ng for serving as my thesis committee. Their comments and feedback to this work are of great value.

I would like to thank Felix Xiaozhu Lin for his great feedback on my research work. I also want to thank Clayton Shepard for the LiveLab data, which is crucial to my research on mobile browsers.

Finally, I want to thank my wife Sijie and my parents for their great love and support. I cannot succeed without them.

Contents

List of Figures

List of Tables

Chapter 1 **Introduction**

Web browsers are among the most important applications on mobile devices including smartphones and tablets, but it is known to be slow, taking many seconds to open a webpage. The long delay harms mobile user experience and eventually discourages web-based business. For example, Google will lose up to 20% traffic with every 500 ms extra delay and Amazon will lose 1% sales with every 100 ms extra delay [25].

Understanding why the mobile 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 [30, 48, 59]. On the other hand, a recent study [20] demonstrates that the wireless hop with its long round-trip time (RTT) can significantly slow down the browser. However, the authors take a black-box approach without looking into the internals of the web browser; thereby they provide limited insights.

We examine the internals of web browsers on mobile devices and make the following key findings. (*i*) Improvement on compute-intensive operations suggested by prior work such as *style formatting*, *layout calculation* [30, 48, 59], and *JavaScript execution* [20] will lead to marginal improvement in browser performance on mobile devices. (*ii*) Instead, *resource loading*, the process that fetches the resources required to open a webpage, is the key to browser performance on mobile devices. (*iii*) Given a resource, the delay of resource loading is determined by the network condition, the browser loading procedure and the processing power of a mobile device. Our results agree with the findings from [20] that a long network RTT is detrimental to the browser performance. We further find that improvement in network bandwidth will not improve browser performance much beyond a typical 3G network. Finally, by comparing the behaviors of two smartphones, Google Nexus One (N1) and HTC Dream (G1), we observe that more powerful hardware will reduce the browser delay mainly by accelerating OS services including the network stack, instead of speeding up the compute-intensive operations suggested by prior work [30, 48, 59].

To speed up mobile browsers by improving resource loading, many effective solutions require infrastructure support, e.g., thin-client approaches [24, 27, 38, 46], session-level techniques [43], prefetching [2, 6, 11, 29, 39] and SPDY, a new network protocol [51]. They are limited in one or more of the following ways. First, solutions requiring web server support are difficult to deploy and may not work for legacy websites. The adoption of a new protocol like SPDY [51] will take a long time, if it ever happens. Second, infrastructure support depends on server or proxy capabilities and does not scale up very well with the number of clients. For example, the failure of Amazon Web Services' cloud-computing infrastructure [37] takes many websites down. Finally, solutions based on proxy support violate end-to-end security, which is crucial to secure websites.

Not surprisingly, solutions that do not rely on infrastructure support, or *client-only* solutions, are particularly attractive because they are immediately deployable, scalable, and secure. There are two orthogonal types of client-only solutions to improve resource loading: hardware improvement and RTT hiding. Hardware improvement on a mobile device makes the network stack and other OS services faster, resulting in faster resource loading in the browser. The hardware improvement is likely to continue because of Moore's Law. RTT hiding tries to hide or save the RTT spent when opening a webpage, e.g., caching and web prefetching. This thesis focuses on the client-only solutions for RTT hiding to speed up mobile browsers without any hardware change on mobile devices.

While client-only solutions are likely to be less effective than those leveraging infrastructure support, it has been an open question how effective client-only solutions can be for mobile browsers. The challenge to answering this question has been the lack of data regarding the browsing behavior of mobile users.

The technical goal of this thesis is to study the speedup of mobile browsers by client-only solutions, with the help of an unprecedented dataset of web browsing data continuously collected from 24 iPhone users over one year, or LiveLab traces [44]. In achieving our goal, this thesis makes five contributions. First, we characterize the performance of mobile browsers and find that the bottleneck is resource loading instead of compute-intensive operations.

Second, we study the browsing behavior of mobile device users and the webpages visited by them. We find that subresources needed for rendering a webpage can be much more predictable than which webpage a user will visit because subresources have a much higher revisit rate and a lot of them are shared by webpages from the same website.

Third, we quantitatively evaluate two popular client-only approaches: *caching* and *prefetching*. *Caching* seeks to store frequently used web resources locally; but we find that it has very limited effectiveness from the LiveLab traces: 60% of the requested resources are either expired or not in the cache. *Web prefetching*, e.g., [6, 11, 39], seeks to predict which webpage is likely to be visited by the user, and then fetches all the resources needed to render the webpage beforehand. While web prefetching with infrastructure support, e.g., [6, 11, 39], is known to be effective by aggregating many users' behavior, we find that, on mobile devices, client-only prefetching is ineffective and harmful because webpages visited by mobile users are less predictable: over 75% of the visits in the LiveLab traces are to webpages visited only once.

Fourth, we propose and study a new, orthogonal client-only approach: speculative loading. Given a web URL, speculative loading leverages concurrent TCP connections available to modern browsers and loads subresources that are likely to be needed, concurrently with the main HTML file. To determine which subresources to load, the browser maps out how a website organizes resources based on the browsing history. We implement speculative loading in a WebKit-based browser called *Tempo* and evaluate it on real mobile devices with a 3G network. The evaluation shows that, on average, Tempo can improve browser speed by 1 second (~20%) with small data overhead. This will not only make web browsing noticeably faster but may also increase traffic to Google by up to 40% and increase Amazon sales by up to 10% according to [25].

Finally, because caching, prefetching, and speculative loading represent the three fundamental approaches that a client can improve resource loading in mobile browsers, our study enables us to find the effectiveness of client-only solutions empirically: the upper bound of the mobile browser delay reduction from client-only solutions is about 1.4 seconds on average for the websites visited by the LiveLab iPhone users. The client-only solutions are limited for four reasons: (*i*) a large portion of web resources are either not in the cache or their cached copies quickly expire; (*ii*) mobile browsing behaviors are not very predictable; (*iii*) a client cannot completely predict what resources are needed for a webpage based on its user's history; (*iv*) the request-response model of HTTP [1] requires at least one request for each resource needed, which magnifies the impact of the relative long RTT of cellular networks. While 1.4 seconds is nontrivial, to make mobile browsers instantly fast, infrastructure support is still necessary.

What Tempo achieves is very close to the upper bound. Tempo can also be combined with infrastructure support by providing the client with knowledge of the server resources. For example, Tempo can help SPDY [51] to solve the race condition problem.

The rest of the thesis is organized as follows. We first introduce the background of the browser architecture, the characterization methodology for mobile browsers' performance and the long-term mobile web usage traces in Chapter 2. Then we characterize the performance of mobile browsers in Chapter 3 and study the characteristics of mobile browsing and webpages in Chapter 4. Afterwards, we investigate the three fundamental approaches available to client-only solutions in Chapter 5. We provide an empirical analysis of the upper bound of improvement possible by client-only solutions. We present the design and implementation of Tempo in Chapter 6 and offer the results from lab and field based evaluations of Tempo in Chapter 7. We discuss the related work in Chapter 8 and finally conclude the thesis in Chapter 9.

Chapter 2 **Background**

We first provide an overview about how a mobile browser works by using WebKit [57] based browsers. Then we discuss the characterization methodology applied to study the performance of mobile browsers. Finally, we introduce the long-term mobile web usage traces we utilize in our study.

2.1 Browser Architecture

A modern browser is a very complicated piece of software. For example, the source code of the WebKit browser engine in Android 2.1 has around one million lines in over 5,700 files [57]. When opening a webpage, a 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 JavaScript 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.

The procedure of opening a webpage, as 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 the local cache. Resource loading uses services from the underlying network stack,

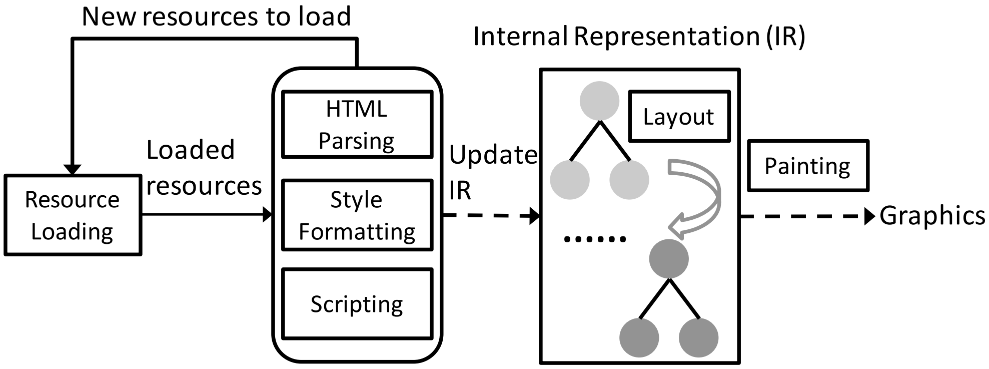


Figure 1: The procedure of opening a webpage

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, the network bandwidth, the 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 webpage. The five operations are *HTMLParsing* (or *Parsing*), *StyleFormatting* (or *Style*), *Scripting*, *Layout*, and *Painting*. The first three operations process HTML documents, style constraints (e.g., CSS), and JavaScript codes, 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 webpage. Finally, all other processing incurred by the browser is treated as one operation, called *Glue* operation, in this thesis.

While it is tempting to think the first six operations described above as a pipeline, three key properties of them make the webpage opening procedure far more complicated. First, when opening a webpage, an operation can be performed many times. For example, it often takes *multiple* loading-processing iterations over *multiple* resources to finish opening a webpage. 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* ongoing 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.

2.2 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 webpage loading progress bar indicates 100%. Such latency covers the time spent in all operations involved when opening a webpage, and can be unambiguously measured by keeping time-stamps in the browser code. Modern browsers utilize incremental rendering to display a partially downloaded webpage to users. We do not consider a partially displayed webpage as the metric because it is subjective how partial is enough to say the webpage is opened.

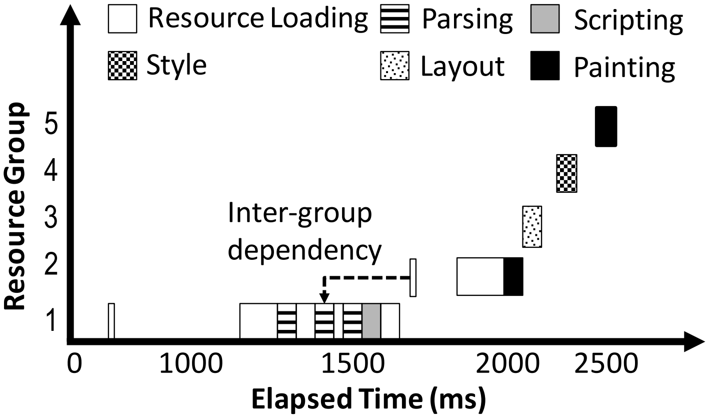


Figure 2: **A simplified dependency timeline graph**

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.

2.2.1 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 2. 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 to be discussed in Chapter 2.2.2. 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 the WebKit browser engine. 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 webpage 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.

The proposed dependency timeline is motivated by the timeline panel [12] provided by the WebKit [57] browser engine. 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.

2.2.2 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 operation instances depending on each instance to the left of the time axis, i.e., to be 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, e.g., Layout or Paint, it will shift the same distance as the most recent IR-producing operation instance does.

2.3 LiveLab: Web Usage by 24 iPhone Users

We leverage web usage data collected from LiveLab [44], an unprecedented study of 24 iPhone 3GS users from February 2010 to February 2011. The 24 participants are recruited with balanced gender, major, and socioeconomic status to represent the Rice University undergraduate population, but not the general user population who use mobile devices. All participants receive unlimited data and are required to use the outfitted iPhone as his or her primary mobile device. Almost all aspects of the participants' iPhone usage and context were collected by a piece of in-device, *in situ* programmable, logging software. The web usage data used in this work contains user ids, timestamps and URLs of webpages visited. The top 10 visited websites by each LiveLab user account for the majority (81%) of the user's webpage visits. Out of the top 10 visited websites of each LiveLab user, there are 94 websites, which will be used as benchmark websites in our study. The LiveLab web usage data provide us a unique opportunity to understand the browsing behavior of mobile users.

The 24 participants obviously cannot represent the general user population who use mobile devices. However, they do provide us an important window into the latter. More importantly, most of our findings are not tied to the special demography of the 24 participants and we believe most, if not all, conclusions drawn in this paper regarding the performance of mobile browsers should be applicable to a large fraction of the general population.

Chapter 3 **Performance Characterization of Mobile Browsers**

As one of the most important applications on mobile devices, the web browser is known to be slow and often takes seconds or tens of seconds to open a webpage. Understanding why the browser is slow on mobile devices is critical to its optimization. We next present our comprehensive study on the performance characterization of mobile browsers and the reasons why mobile browsers are slow. Our study shows that the bottleneck of mobile browser performance is in resource loading instead of several compute intensive operations, e.g., Style, Layout and Scripting, as suggested by prior work [30, 48, 59].

3.1 Experimental Setup

We first describe the experimental settings as follows.

Mobile Device 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 1 GHz Qualcomm Snapdragon Application Processor while G1 has a 528 MHz Qualcomm MSM7201A Application Processor. Both smartphones run identical software stacks: the Android 2.1 operating system with our instrumented WebKit browser engine.

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 1 Gbps 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 negligible impact on the network performance: the average RTT between the smartphone and the gateway's Ethernet interface is 1 ms; the forwarding bandwidth provided by the gateway is 54 Mbps, 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 [36] is 23 ms. To examine the impact of the network RTT and the 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 a typical 3G network, 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 [36] is 276 ms 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 [36]. The *non-mobile* set consists of the 10 most visited non-mobile webpages from the first three months' data of LiveLab traces [44]. These webpages were visited 2611 times during the three months.

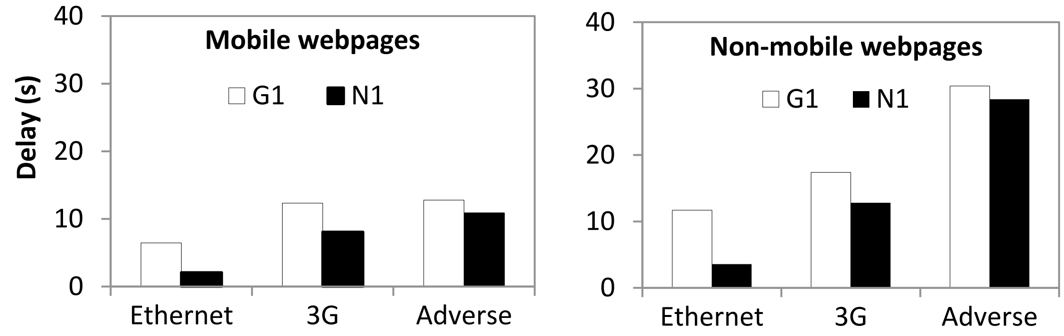


Figure 3: Average browser delay for opening mobile and non-mobile webpages on G1 and N1 through three different networks. Adverse network is emulated with 400 ms injected delay in the RTT and 500/100 Kbps downlink/uplink bandwidth

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 [50] on the smartphone to record the network traffic, e.g., TCP packets, when opening a webpage. According to our measurements, the overhead of tcpdump is negligible (<2% of CPU time and <0.4% of memory).

3.2 Characterization Results

We next present findings from the characterization study. Not surprisingly, mobile browsers are slow, even for mobile webpages. Figure 3 presents the average browser delay on N1 and G1 under three different network conditions for two benchmarks. Adverse network is emulated with 400 ms injected delay in the RTT and the 500/100 Kbps downlink/uplink bandwidth [20].

(*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 chapter, 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 Chapter 2.2.2 to evaluate the impact of accelerating browser operations in various ways. Our results highlight the limitations of prior work on browser performance characterization.

3.3 IR Operations Do Not Matter Much

Prior work on browser performance characterization and optimization suggests that optimizing some of the IR operations would be profitable because they are compute-intensive. For example, the IE8 team [48] focused on Layout and Painting; the authors of [30] focused on Layout; and the authors of [59] 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

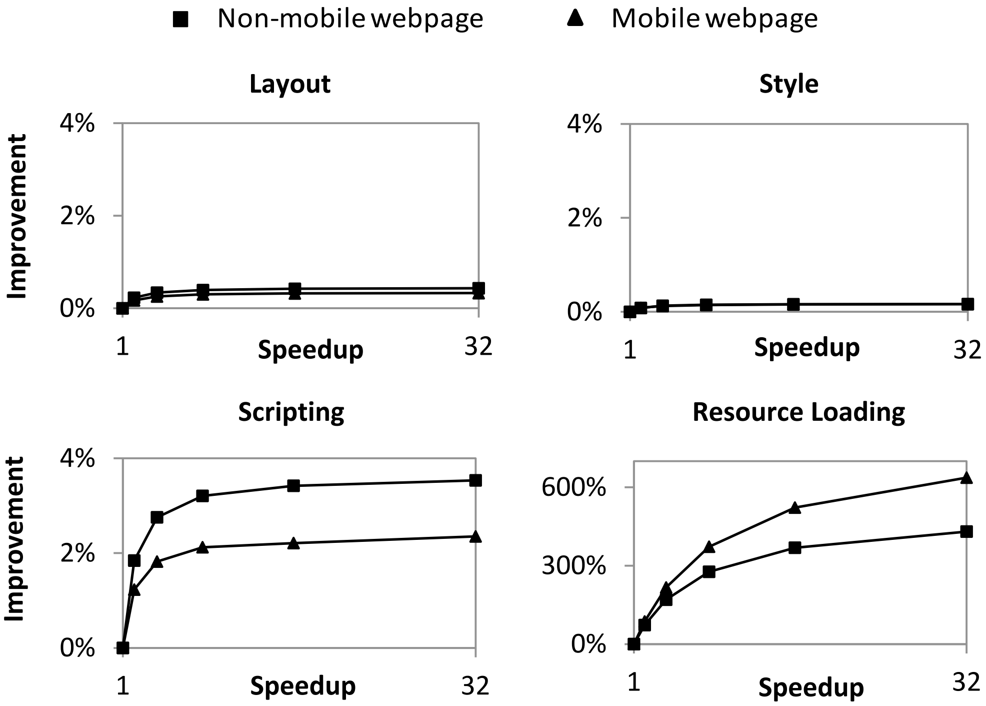


Figure 4: Browser performance improvement when speeding up various operations using a typical 3G network (N1)

that improving these IR operations will only lead to marginal browser delay improvement.

Figure 4 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 32-fold speedup of Layout, Style, and Scripting, the browser performance will only be improved by 0.4%, 0.2%, and 4%, respectively. Note that 32-fold speedup would require significantly advancement in hardware and algorithm. For example, the JavaScript engine V8 [14] introduced in Android 2.2 can improve scripting by only 2-3 times [16], resulting in 3% browser performance improvement according our analysis. This result also indicates that the JavaScript benchmark used by [20] may not be representative of JavaScript codes encountered by mobile browsers in the field. With Ethernet, the overall improvement from speeding up computing is higher than that with a typical 3G network as expected. Yet the conclusions drawn from a typical 3G network still hold. With 32-fold 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. 32-fold speedup of all five IR operations improves the browser performance by 7% with 3G and by 23% with Ethernet. 32-fold speedup of all five IR operations plus the glue operation improves the browser performance by 8% with 3G and by 27% with Ethernet.

3.4 Resource Loading Rules

What-if analysis reported in Figure 4 demonstrates that the source of the browser performance problem is in resource loading: Twofold speedup of resource loading will improve the browser performance 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 Chapter 2.1, 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, which are determined by the resource files' types. 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, the resource loading procedure, and the processing power available on the mobile device. We next examine how these four factors impact the overall browser delay.

3.4.1 Network RTT and Bandwidth

Using the Ethernet setup described in Chapter 3.1, we inject various RTT delays and set assorted bandwidth limits to emulate the impact of the network RTT and bandwidth. While varying one of the two metrics, we fix the other to their typical values in the 3G network [20]: a RTT of 200 ms, and a bandwidth of 1000 Kbps downlink and 200 Kbps uplink. The injected RTT varies from 0 ms to 400 ms. The downlink/uplink bandwidth limit varies from 250/50 to1500/400, in Kbps. Figure 5 presents the measurement results.

We make the following observations.

(*i*) Improving the bandwidth does not improve the browser delay much after 1000/200 Kbps for downlink/uplink, as shown in Figure 5. Therefore, current typical 3G networks can be considered as adequate since their throughputs are usually in this level or much higher [20]. This is not surprising because the size of all resources for a webpage is usually a few hundred KB according to our measurements: 160 KB and 770 KB for mobile

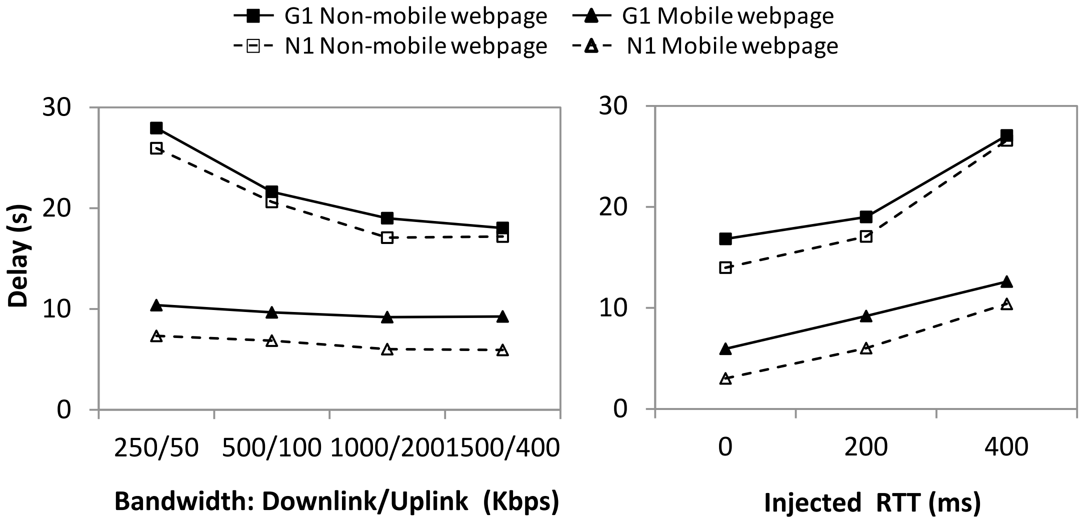


Figure 5: Impact of the bandwidth limitation and the network RTT on browser delay for G1 (solid lines) and N1 (dashed lines)

and non-mobile webpages, respectively. However, as webpages are likely to become richer and therefore come with larger resource files in the future, bandwidth improvement will certainly help.

(*ii*) The network RTT is a key factor to the browser delay as also observed by the authors of [20]. As shown in Figure 5, the browser delay increases significantly when the injected RTT increases from 0 ms (Ethernet) to 200 ms (typical 3G) to 400 ms (adverse 3G). The findings regarding the impacts by the bandwidth and the RTT imply that the browser delay difference between the Ethernet and the typical 3G network, as shown in Figure 3, should be attributed to the difference in the RTT rather than that in the bandwidth.

3.4.2 Resource Loading Procedure

The 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 the browser is parsing a loaded resource, e.g., the main HTML file. (*ii*) Redirections on the main HTML file further delay the discovery of later resources. (*iii*) If JavaScript is used, the parsing of the HTML file will be blocked until the JavaScript code has been executed. A side parser [26] can execute the JavaScript code without blocking HTML parsing, but it is not yet widely used. (*iv*) Finally, the limited number of concurrent TCP connections and the sequential secure connection (HTTPS) establishment further serialize the loading of multiple resources.

Loading a resource incurs multiple network round trips in series, due to redirection, DNS query, TCP connection establishment, secure connection creation and resource file downloads. Typically, loading incurs 3 round trips for HTTP files and 5 for HTTPS ones. The exact number of round trips varies according to the real situations. For example, redirections and TCP slow start will increase the number of round trips; and domain name pre-resolution and TCP connection reuse will reduce the number of round trips. Under the current resource loading procedure, on average 18.6 and 27.2 round trips are incurred in series for opening a webpage from top 10 mobile websites and from top10 non-mobile

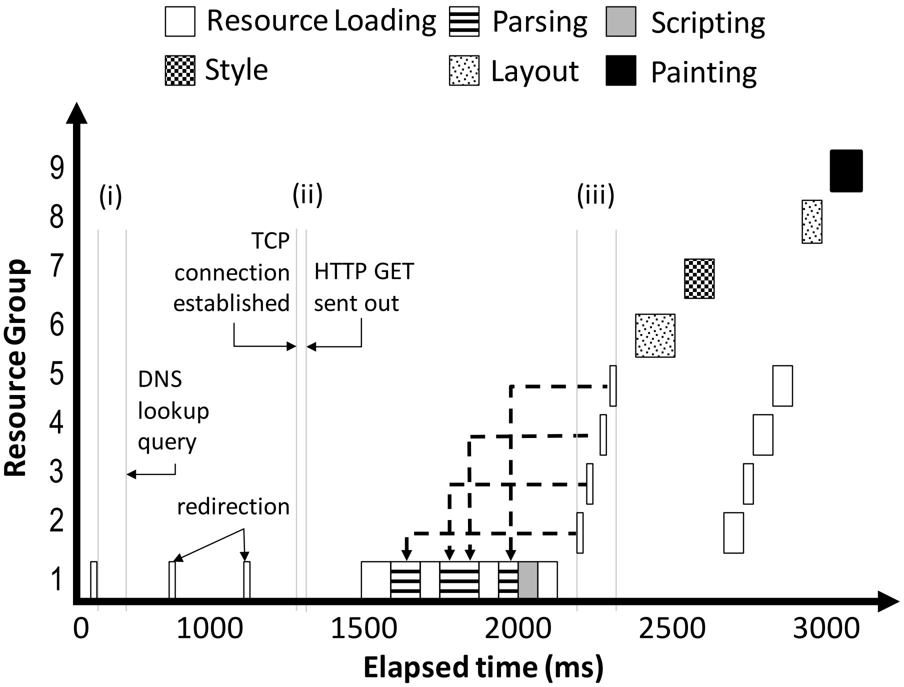


Figure 6: Dependency timeline graph for opening mail.yahoo.com with Ethernet network condition on G1. (i) is incurred for each resource; (ii) is incurred for each network round trip; (iii) is incurred when multiple resources are requested at the same time.

websites, respectively. Such large numbers of round trips in series are the key reason that the network RTT matters a lot to the browser delay, which is discussed in Chapter 3.4.1.

3.4.3 Processing Power

The resource loading time also depends on the processing power available on the mobile device since it involves the network stack and other OS services on the mobile device. A careful examination of the dependency timeline graphs reveals three categories of time intervals in resource loading in which N1 significantly outperform G1. Figure 6 illustrates these three categories for mail.yahoo.com.

(i) Time between when SendResourceRequest is made by the WebKit browser engine and when the first 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 including the network stack are invoked. This process adds delay to the loading of every resource.

(ii) Time between when the TCP connection for a resource request is established and when the HTTP GET is sent out. During this time, the OS notifies the browser when the connection is up; the browser then invokes the network stack to send out the HTTP request. This process adds delay to every round trip into the network.

(iii) Time spent sending 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 service, e.g., domain name resolution, TCP connection establishment, and packet transmission. This time is incurred when multiple resources are requested at the same time.

Apparently the time intervals of category (*ii*) are much more frequent but shorter than those of category (*i*). Figure 7 presents the time G1 and N1 spent in each of the three categories when 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 time spent in the three categories of intervals will be 1.1 seconds on average for N1 and 2 seconds for G1 when opening a mobile webpage.

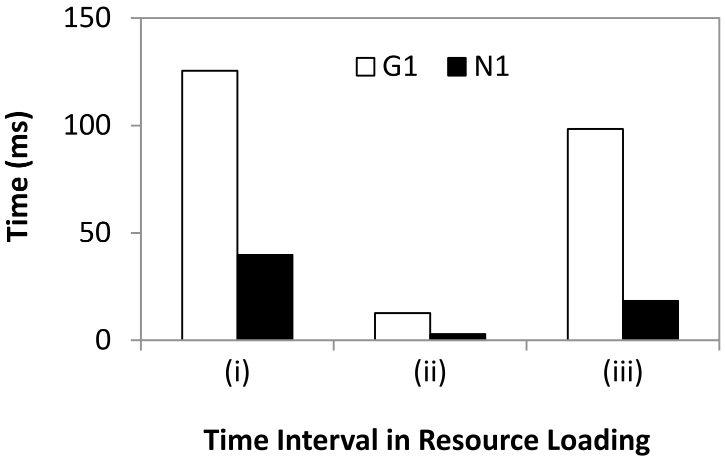


Figure 7: Time spent by G1 and N1 for three time intervals in resource loading

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

3.5 Summary

In this chapter, we show that the bottleneck of mobile browser performance is in resource loading due to the long RTT, the large number of total round trips resulting from inefficient resource loading procedure, and the slow OS services including the network stack. In contrast, speeding up compute-intensive operations will only lead to marginally improvement in browser performance.

The characterization results suggest three ways to improve resource loading: reducing the RTT, hiding the RTT and improving hardware. RTT reduction requires infrastructure support, which is hard to deploy. In contrast, RTT hiding and hardware improvement can be realized without infrastructure support. While hardware improvement is likely to continue because of Moore’s Law, this thesis focuses on client-only solutions for hiding the RTT to speed up mobile browsers without hardware change, e.g., caching and web prefetching to be discussed in Chapter 5.

Our observations regarding the inefficient resource loading procedure motivate us for new client-only solutions to hide the RTT. As discussed in Chapter 3.4.2, subresources can only be discovered and requested after the main resource is downloaded and parsed. If redirection occurs, the process will be much longer. On mobile devices, loading the main resource, e.g., an HTML file, can contribute more than 50% of the browser delay. On average, getting the first data packet of the main resource takes 2 seconds under 3G network. If the main resource contains JavaScript code, the parsing of the main resource file can be further delayed, resulting in even longer time to discover subresources. Moreover, the dependencies between the resources will further serialize the resource loading operations [28]. Those observations motivate our proposed client-only solution, speculative loading, to fully parallelize resource loading of mobile browsers, as will be discussed in Chapter 5.3.

Chapter 4 **Mobile Web Browsing Characteristics**

To study the effectiveness of client-only solutions for mobile browsers, we first study the web browsing behavior of mobile users and then characterize visited websites by using the LiveLab traces.

4.1 Characteristics of Websites

Since resource loading is the key to the performance of mobile browsers, we can gain insight for improvement by examining how a webpage needs many resources and how webpages from a website share resources. Toward this end, we represent each website, its subdomains, webpages, and subresources with a graph, called the *resource graph*. Figure 8 shows an example of the resource graph for the simplified Rice University website. A resource graph has four types of nodes: *website node*, *subdomain node*, *webpage node* and *subresource node*. A *website node* is represented by the top two level domain names of the website. A *subdomain node* is a subdomain of the website. *Webpage* and *subresource nodes* are the real resources in the website and can be addressed by their URLs. The webpages mainly correspond to HTML files and the subresources mainly correspond to JavaScript, CSS, and image files.

The arrows between nodes in a resource graph denote the dependency relationship between a webpage node and its corresponding subresource nodes. That is, subresources can only be discovered after the main resource, i.e., the webpage node, is parsed. Most of the dependencies occur between the webpage node and its subresource nodes. After pars-

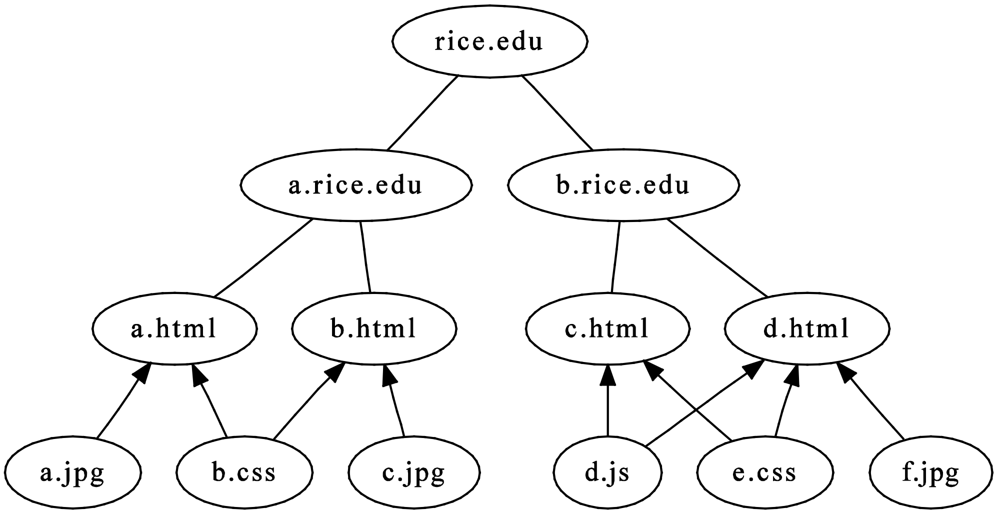


Figure 8: Resource graph for the simplified Rice University website. The arrows correspond to the dependency relationship between the webpage node and the subresource node, i.e., subresources can only be discovered after the main resource is parsed.

ing and executing some JavaScript and CSS files, the browser may discover and request new subresources. With the complete resource graph of a website, we know which subresources are needed to open a webpage of the website.

While each website has its own complete resource graph, a user usually can only see part of it, depending on which webpages the user visited before. We download the homepages of each LiveLab user's top 10 visited websites together with their linked webpages, and then construct a partial resource graph for each website. Though a constructed resource graph is partial, we manually verify that it represents the resource structure of the corresponding website. We have the following two observations.

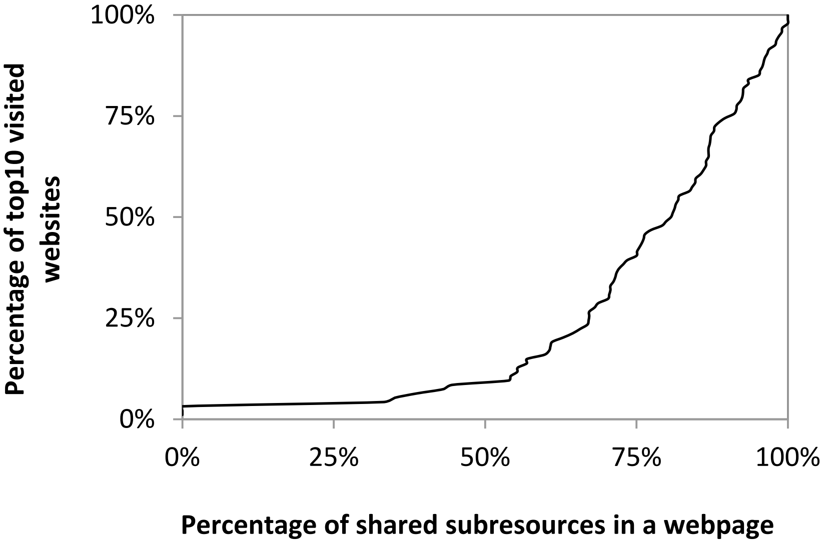


Figure 9: CDF for the average percentage of shared subresources in a webpage, i.e., subresources that are also needed by other webpages in the same website, for 94 websites among the top 10 visited websites of each LiveLab user

First, *webpages from the same website often share a large portion of resources*. In a resource graph, those shared resources are the subresource nodes with multiple outgoing arrows pointing to multiple webpage nodes. Figure 9 shows the cumulative distribution function (CDF) for the average percentage of shared subresources in a webpage, i.e., subresources that are also needed by other webpages in the same website, for the top 10 visited websites. On average, 76% of the resources in one webpage are shared by at least one other webpage from the same website. This observation provides us with a key opportunity to improve the speed of opening a new webpage. After a user visits a website enough times and the resource graph is constructed, the browser can potentially predict the majority of the subresources needed for a new webpage visit, and thus speculatively loads them. The details of subresources prediction for a new webpage visit are discussed in Chapter 6.2.

Second, *the structure of a resource graph can change over time*. New nodes can be added into the resource graph. A typical example is a news website, which has changing content all the time. In addition, resource graphs of different websites change with different frequencies. For each LiveLab user's top 10 visited websites, in total 94 websites, 24 websites add new webpage nodes every few hours or with even shorter periods (fast changing); 13 websites add new webpage nodes daily; and 57 websites are stable and no new webpage nodes are added over a long period of time. Among the fast changing websites, 4% of the webpage nodes and 10% of the subresource nodes are replaced by new ones hourly. Among the unchanged webpage nodes in fast changing websites, 26% of them have new subresource nodes, of which 11% of those subresource nodes are replaced with new ones. This observation challenges solutions that leverage the resource graph, because temporal changes of a website's resource graph are hard to be captured by the client timely. However, our speculative mobile browser design, Tempo, can deal with the temporal changes well and reduce the browser delay by one second. We will evaluate Tempo comprehensively in Chapter 7.

4.2 Browsing Behavior of Mobile Users

Understanding the browsing behavior of mobile device users helps us to study the effectiveness of client-only solutions and better design Tempo. We have four interesting findings. First, *for a given mobile device user, the total number of frequently visited web-*

sites is usually small. On average, each user’s top 10 visited websites account for 81% of his or her total webpage visits. Therefore, it is reasonable to focus on the resource loading optimization for the webpages that belong to the top 10 visited websites.

Second, mobile users vary significantly in their web usage. Approximately three (both average and median) of each user’s top 10 websites are shared by the all-users-combined top-10 visited websites. Therefore, resource loading optimization should target different sets of websites for different users, which can be easily achieved by client-only solutions.

Third, *the majority of webpage visits are new visits*. On average, 75% of the webpages visited are new visits. The high new webpage visit rate is one of the reasons that client-only web prefetching has poor performance on mobile browsers. We will evaluate client-only web prefetching in details in Chapter 5.2.

Fourth, *though users tend to visit new webpages, a mobile browser is likely to request a similar set of subresources.* On average, only 35% of the subresources requested are new subresources. The reason is that webpages in the same website share subresources, as discussed in Chapter 4.1. Therefore, subresources can be much more predictable than webpages. This is the key reason that Tempo outperforms client-only web prefetching.

Chapter 5 **Effectiveness of Client-Only Approaches**

Driven by findings presented in Chapter 4, we next examine three orthogonal client-only approaches that speed up resource loading. With *caching,* a browser saves the subresources of previously visited webpages locally and reduces the resource loading time if the same subresources are requested again. *Web Prefetching* predicts which webpage a user is likely to visit next and downloads its resources beforehand; it minimizes the resource loading time if the user does visit a prefetched webpage. We show both caching and prefetching are limited for mobile browsers, and show how a new, orthogonal approach, called *speculative loading*, can be much more effective.

5.1 Caching

Caching is a well-known approach used to fight I/O bottlenecks. A browser stores frequently used web resources locally to save RTT and bandwidth. But resources with "no-store" specified in the cache-control header field cannot be stored in the browser cache.

A cached resource can have two states: fresh or expired. The browser can return a fresh resource in response to the request without contacting the server. The browser needs to revalidate an expired resource with the origin server to see if the resource is still usable. If it is usable, the server will not send back the resource file. Resources with "no-cache" specified in the cache-control header field can be actually cached, but they immediately expire. Both HTTP and HTTPS resources can be cached. Their expiration time can be indicated from their header fields specified by the server.

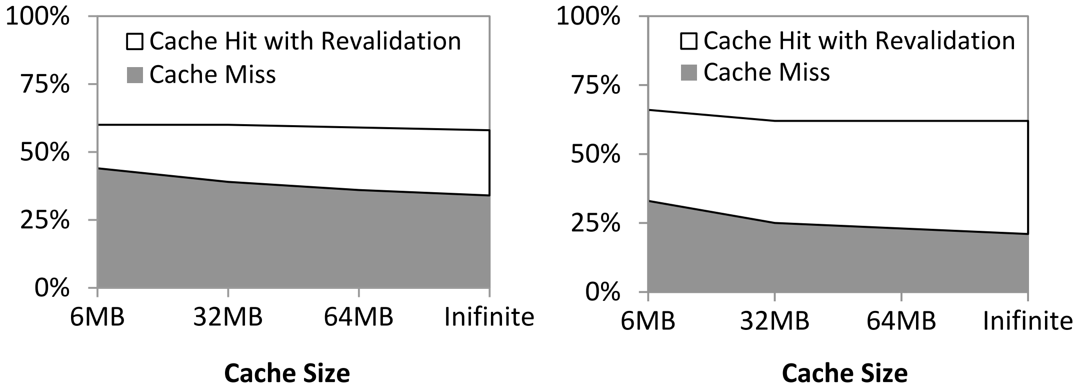


Figure 10: Cache simulation results for the webpages from all websites (Left) and top 10 visited websites (Right) in the LiveLab traces

A realistic browser cache is a mixture of fresh and expired resources. Because a large portion of mobile web resources either cannot be cached or have a short expiration time, caching brings little benefit to mobile browsing. Usually, by revalidating expired resources with the origin server, a browser avoids re-fetching resources if their local copies are still usable. However, revalidations cannot hide the extra network RTT, and the RTT is the most important factor to the mobile browser delay [56]. As a result, latencies in revalidations make caching ineffective for mobile browsers.

We experimentally show how excessive revalidations outweigh the benefit of caching with the LiveLab traces [44]. First, we download all the resources of the webpages with header fields. Then we simulate the cache behavior of a mobile browser by replaying each LiveLab user's browsing history using the resource files and their header fields. We repeat the simulation with four cache sizes: 6 MB, 32 MB, 64 MB and infinite. Note that the cache size of Android Gingerbread's default browser is 6 MB. We have to exclude about 32% of the webpage visits, including visits to webpages that no longer exist (39%)

and to HTTPS webpages (61%) that require users' login information. Excluding HTTPS webpages does not bias the results much because most of their resources' expiration time is not different from their HTTP counterpart.

As shown in Figure 10, our simulation results show that 60% of resource requests incur network activities with a 6 MB cache. Network activity is required when a requested resource is not in the cache (cache miss). It may also be required even if the resource is in the cache: when a resource being requested is cached but expired, the browser still has to contact the origin server to revalidate it. As is apparent from Figure 10, the effectiveness of caching is even lower for the top 10 visited websites of each user: 70% of resource requests incur network activities, and half of the activities are due to revalidations.

Note that increasing the cache size will not help much. Figure 10 shows that a small browser cache (6 MB) only incurs 10% more cache misses than an infinite size cache. And 58% of resource requests still incur network activities with an infinite size cache, which is close to the percentage with a 6 MB cache (60%). Therefore, a larger cache will not bring much more benefit.

In summary, our results show that the benefit of caching is marginal because it can do little in loading resources whose cached copies expire quickly: revalidations save bandwidth usage in this case, but cannot hide network RTT, which is the most important factor to the performance of mobile browsers [56].

5.2 Web Prefetching

We believe that client-only web prefetching [6, 11, 39] is harmful to mobile web browsing because it results in significant additional data usage with very little speed improvement. Web prefetching predicts the webpages that will be visited by the user and downloads their resources beforehand. When the user actually visits a predicted webpage, its resources are already available locally. Most solutions of web prefetching are intended for PC browsers and involve infrastructure support to aggregate behavior of many users. We show that on mobile devices, client-only web prefetching is ineffective because it cannot predict URLs that have never been visited before. On average, 75% of the webpages visited by LiveLab users are new visits, as shown in Chapter 4.2, leading to low accuracy of webpage predictions.

To quantitatively demonstrate its ineffectiveness, we evaluate client-only web prefetching by using the LiveLab traces. We simulate the web prefetching algorithm presented in [6], called *most-popular*. It uses the popularity ranking of a user's past webpage requests to predict future webpage requests. We also borrow the metrics, *hit ratio* and *usefulness*, from [6]. The *hit ratio* is defined as the number of webpages that are predicted and also actually requested to the number of predicted webpages. It represents the accuracy of the prediction. High hit ratio means low unnecessary data usage. The *usefulness* is defined as the number of webpages that are predicted and also actually requested to the number of actually requested webpages. It represents the coverage of the prediction. High usefulness means high average speedup.

With a one-month training period, the hit ratio is 16% and the usefulness is 1% on average among 24 LivaLab users. Such low hit ratio and usefulness lead to considerable unnecessary data usage yet very limited speed improvement of mobile browsers. With a very generous assumption that the prefetched content is cached and will not expire before the actual visit, the upper bound of the reduction of the mobile browser delay from the most-popular web prefetching algorithm is 1%. And the unnecessary data usage accounts for 84% of the total prefetched data. With a different training period, the client-only web prefetching will still be ineffective because of the large amount of new webpage visits.

One may think that prefetched subresources for one webpage may help in loading other webpages from the same website faster because subresources are shared by webpages from the same website, as shown in Chapter 4.1. Unfortunately, this is usually not the case because many resources are either not in the cache or their cached copies expire quickly, as shown in Chapter 5.1. In contrast, speculative loading solves this problem by loading the resources only *after* a user requests a webpage's URL.

5.3 Speculative Loading

Seeing the failures of caching and prefetching, we propose a third, orthogonal approach called *speculative loading* that loads subresources for a webpage concurrently with the main resource file *after* a user provides the web URL.

Essentially, speculative loading predicts which subresources to load based on a resource graph of the website constructed using knowledge of the website collected from the past. It leverages the multiple concurrent TCP connections, e.g., four for Android

Gingerbread’s default browser, available to the browser to concurrently load subresources with the main resource. Unlike caching, speculative loading will revalidate expired resources and load evicted resources concurrently while loading the main resource, thus keeping most subresources fresh in cache when the browser actually requests them.

Figure 11 illustrates the difference between the legacy loading and speculative loading. Unlike web prefetching, speculative loading predicts which resources a webpage may need, instead of which webpages a user may visit.

5.3.1 Upper Bound of Improvement

The key to the effectiveness of speculative loading is *subresource prediction*. By assuming 100% hit ratio and 100% usefulness for subresource prediction, one can derive the upper bound of the browser delay reduction from speculative loading. We will show in Chapter 7.2 that the performance of speculative loading is close to this upper bound in practice. Here we examine the browser delays for the homepages of top visited websites from LiveLab traces under three different cache states: fresh, expired, and empty. With a fresh cache, the browser will use the cached copy of a requested resource without any network activity. With an expired cache, the browser needs to revalidate a cached resource with the origin server. With an empty cache, the browser needs to load every resource from the server.

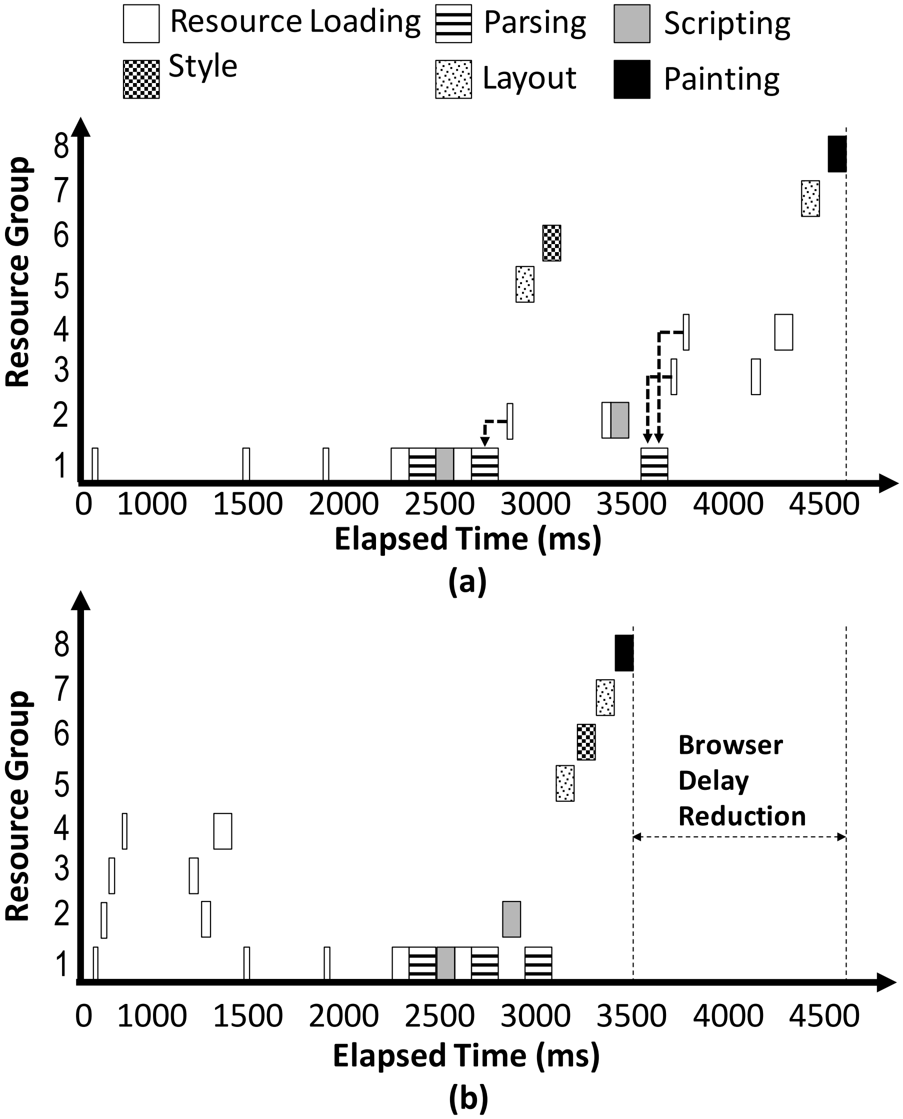


Figure 11: Dependency timeline graph for (a) legacy resource loading and (b) speculative resource loading. Speculative loading eliminates the dependency relation and loads main resource and subresources simultaneously. It also eliminates the delay due to the execution of JavaScript codes.

Table 1 shows the upper bound of the browser delay reduction. We measure the browser delays of legacy loading with an empty cache on Samsung Galaxy S II in 3G network provided by U.S. wireless carrier AT&T. Then we simulate the browser delays in other columns by applying what-if analysis, which is discussed in Chapter 2.2.2.

We have three observations. (*i*) The average browser delay reduction is 33% (~2 seconds) for an expired or an empty cache. The reduction comes from the time waiting for the main resource to discover its subresources. (*ii*) There is nearly no reduction for a fresh cache because all the subresources are available locally already. There is no advantage of discovering and loading subresources speculatively. (*iii*) The upper bound of the browser delay reduction by speculative loading for a realistic cache can be estimated to be around 1.4 seconds (22%) because, when a webpage from top10 visited websites is visited, 70% of its subresources needed by a webpage are either expired or not in the cache, as shown in Chapter 5.1.

5.3.2 Predicting Server vs. Predicting User

Speculative loading shows more promise than web prefetching. The upper bound of the browser delay reduction from speculative loading (22%) is one order of magnitude larger than the upper bound of reduction from web prefetching (1%). Moreover, by applying the design discussed in Chapter 6, speculative loading will incur much lower overhead of wireless data usage with 65% hit ratio as will be evaluated in Chapter 7.1, comparing to 16% hit ratio for web prefetching.

Table 1: Upper bound of the browser delay reduction from speculative loading under different cache states (in ms)

	Sites	Legacy	Speculate	Reduction	
Empty Cache	ESPN	7143	4622	2521	35%
	CNN	6300	4315	1985	32%
	Google	3661	2223	1438	39%
	Yahoo! Mail	4341	3199	1142	26%
	Weather	6349	3608	2741	43%
	Craigslist	3103	1920	1183	38%
	Neopets Games	11843	9340	2503	21%
	Varsity Tutors	9219	7405	1814	20%
	Ride METRO	8774	6068	2706	31%
	Rice Registrar	6427	3541	2886	45%
	Average	**6716**	**4624**	**2092**	**33%**

	Sites	Legacy	Speculate	Reduction	
Expired Cache	ESPN	6702	4622	2080	31%
	CNN	4869	2884	1985	41%
	Google	3363	2131	1232	37%
	Yahoo! Mail	4333	3199	1134	26%
	Weather	6294	3608	2686	43%
	Craigslist	3034	1920	1114	37%
	Neopets Game	11505	9002	2503	22%
	Varsity Tutors	8410	6596	1814	22%
	Ride METRO	8266	5560	2706	33%
	Rice Registrar	5865	3541	2324	40%
	Average	**6264**	**4306**	**1958**	**33%**

	Sites	Legacy	Speculate	Reduction	
Fresh Cache	ESPN	4557	4557	0	0%
	CNN	2382	2382	0	0%
	Google	2162	2131	31	1%
	Yahoo! Mail	3199	3199	0	0%
	Weather	3645	3608	37	1%
	Craigslist	1926	1920	6	0%
	Neopets Games	3605	3605	0	0%
	Varsity Tutors	3313	3313	0	0%
	Ride METRO	3826	3826	0	0%
	Rice Registrar	3351	3351	0	0%
	Average	**3197**	**3189**	**7**	**0%**

There is a fundamental reason that speculative loading can be much more effective than web prefetching: predicting server behavior is much easier than predicting user behavior. Speculative loading predicts the subresources needed by a webpage, which is server behavior prediction. Web prefetching predicts the next visited webpage by the user, which is user behavior prediction. Server behavior prediction can achieve high accuracy because webpages in the same website share subresources, as discussed in Chapter 4.1. On the other hand, user behavior prediction is limited because 75% of the visited webpages are new visits, as presented in Chapter 4.2. To predict server structure, a browser needs to map the resource graph of each website on the mobile device and we will discuss the detailed design in Chapter 6.

5.4 Upper Bound for Client-Only Solutions

Existing two client-only approaches are limited because of two reasons, respectively. First, many mobile web resources are either not in the cache, or their cached copies quickly expire, which makes caching ineffective. Second, behavior of mobile browsing is not very predictable, which makes client-only web prefetching harmful. Our proposed approach, speculative loading, addresses those two limitations by speculatively revalidating expired resources and loading evicted resources, and by predicting server behavior instead of predicting user behavior.

Speculative loading has reached the upper bound of improvement for client-only solutions, i.e., 1.4 seconds as shown by us empirically. The reason is that the request-response model of HTTP protocol [1] requires at least one request for each resource needed and the loading procedure is already fully parallel with speculative loading. In practice, it is difficult to completely predict what resources are needed for a webpage based on a user's history. Our speculative mobile browser design, Tempo, can reduce the browser delay by 1 second, as will be evaluated in Chapter 7, a result close to the upper bound.

As discussed in Chapter 3.5, hardware improvement can also speed up resource loading by providing faster OS services including the network stack. Hardware improvement is orthogonal to the client-only approaches discussed above, which try to hide or save the RTT when opening a webpage. While hardware improvement is likely to continue because of Moore's Law, this thesis focuses on the client-only solutions for RTT hiding to speed up mobile browsers.

Chapter 6 **Tempo: A Speculative Mobile Browser**

We now describe *Tempo*, our mobile browser design that seeks to realize the potential of speculative loading. As illustrated in Figure 12, Tempo is realized by adding a module under the middle layer in Android Gingerbread's default browser. The middle layer connects the WebKit browser engine [57] and the network service provided by the smartphone. It also handles the communication between the browser user interface and the WebKit browser engine, and manages caches, cookies and plug-ins.

Tempo has four components. *Metadata repository* stores each website's resource graph, in particular the dependency information, to make speculative loading possible. S*peculative loader* predicts the needed subresources based on the information provided by metadata repository and loads the predicted subresources speculatively for every webpage visit. *Update service* updates *metadata repository* with the new resource information after a webpage is open and trims the stale nodes in *metadata repository*. The last component is *temporary cache*, which stores resources that cannot be stored in the cache temporarily, i.e., those with "no-store" in the cache-control header filed. We will discuss the details of each component as follows.

6.1 Metadata Repository

Metadata repository is a key-value store, as shown in Figure 13. The key is the website and the value is the website's resource graph, which are discussed in Chapter 4.1. Each

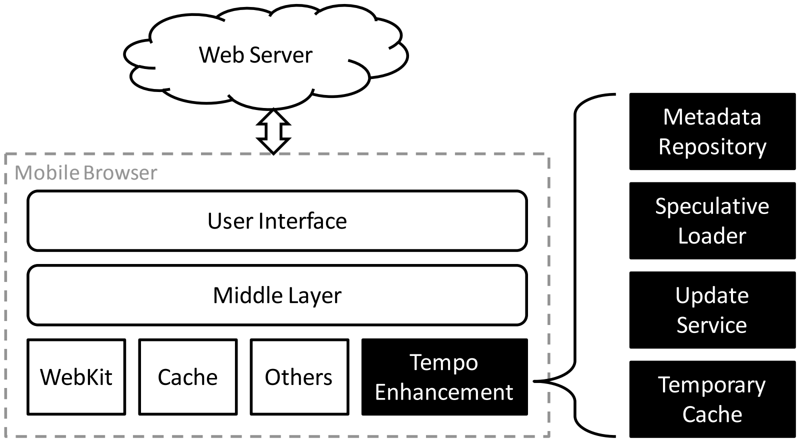
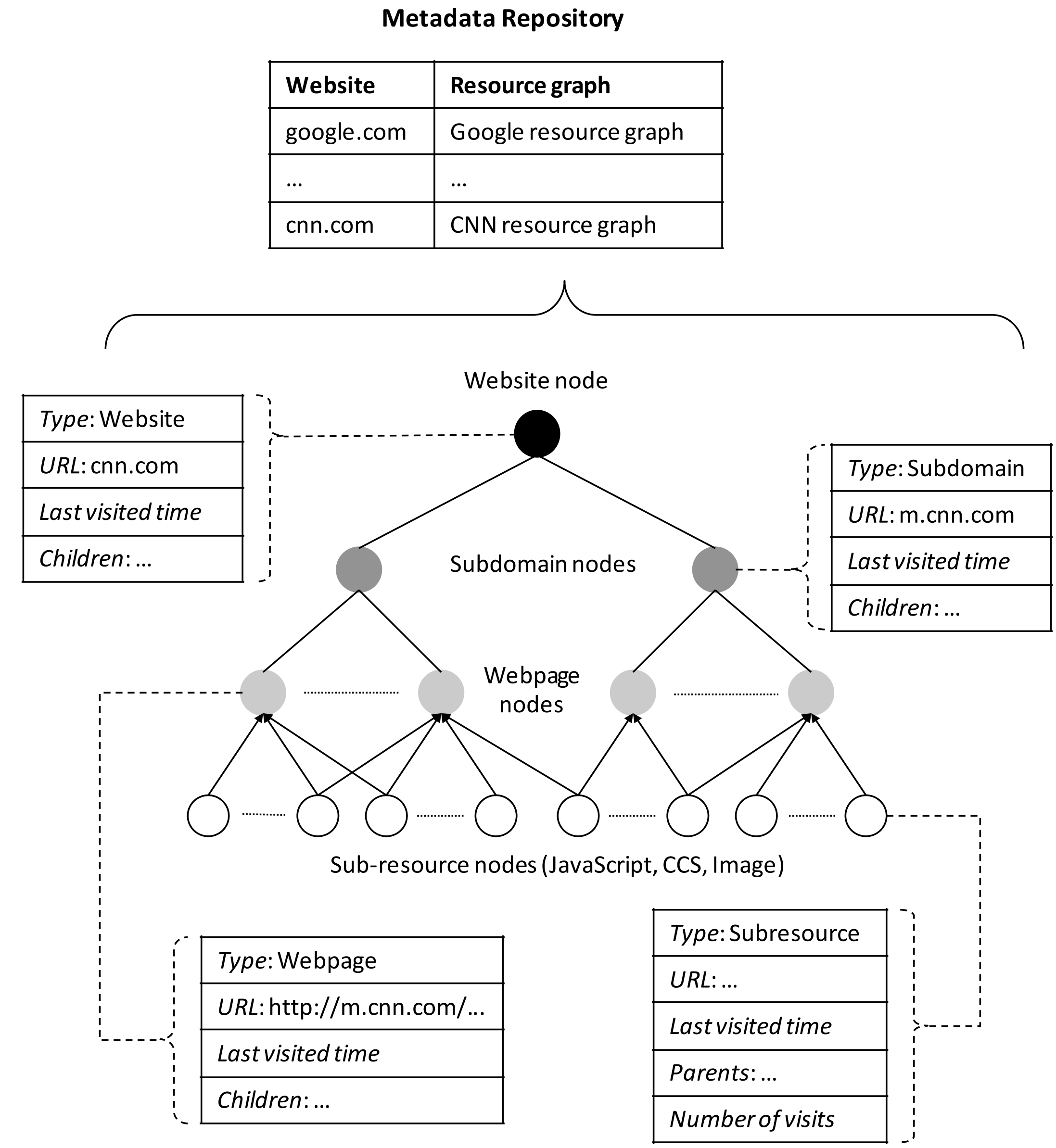


Figure 12: Tempo, a speculative mobile browser. Black components are new additions to existing mobile browsers

node in the resource graph has several fields, e.g., type, URL, last visited time, children, parents, and number of visits. The actual content is not stored in the resource graph.

Our design of metadata repository has two advantages. First, it relates the resources in a website with a resource graph. When a webpage is requested, the browser knows which subresources are needed even before downloading and parsing the main resource file. This makes speculative loading possible. In contrast, caching provides no relation information among the cached resources. Second, metadata repository takes little storage on a mobile device (only several hundred KB) because each node in the resource graph is represented by the URL instead of the actual content.

Metadata repository is stored in the flash storage of mobile devices. Accessing the repository will not affect the browser delay because it will be loaded into the memory when the browser is started and will be saved to the flash storage after each webpage is open.



Metadata Repository

Website	Resource graph
google.com	Google resource graph
…	…
cnn.com	CNN resource graph

Type: Website
URL: cnn.com
Last visited time
Children: …

Type: Subdomain
URL: m.cnn.com
Last visited time
Children: …

Type: Webpage
URL: http://m.cnn.com/…
Last visited time
Children: …

Type: Subresource
URL: …
Last visited time
Parents: …
Number of visits

Figure 13: Metadata repository, a key-value store where keys are websites and values are websites' resource graphs

6.2 Speculative Loader

Speculative loader takes a webpage's URL as the input right after the user enters or clicks the URL of that webpage, retrieves the corresponding resource graph from the metadata repository, predicts the subresources needed for that webpage, and loads those subresources speculatively if they are not in the cache or expired. It handles both webpage revisits and new webpage visits. Note that speculative loading for new webpage visits is very important and cannot be ignored for mobile browsers, because a large portion of webpage visits are new visits, as discussed in Chapter 4.2. In contrast, web prefetching relies on the past history and cannot benefit new webpage visits. This is one reason that web prefetching performs poorly on mobile devices.

The detailed subresource prediction algorithm is illustrated in Figure 14. If the webpage visit is a revisit, speculative loader can find the corresponding webpage node in the resource graph and thus all its child subresource nodes are the subresources the webpage needs. If the webpage visit is a new visit, no corresponding webpage node is stored in the resource graph yet. Speculative loader can predict the subresources' URLs according to the shared subresource nodes because subresources are heavily shared across multiple webpages from the same website, as discussed in Chapter 4.1.

To maximize the prediction accuracy and coverage, speculative loader judiciously prioritizes the candidate subresources by sorting them according to their number of parents (large to small), file types (JS to CSS to image), number of visits (large to small) and URL length (short to long), as indicated by the function `sort()` in Figure 14. If it is a new webpage visit, speculative loader only chooses the ones with high priority as the

```
Input: webpage URL
Output: predicted subresources' URLs
SubresourcePrediction(url):
  candidates = []           // subresources
  webpage_node = get_webpage_node(url)
  if webpage_node != NULL:  // webpage revisit
    candidates = children_of_webpage_node
    sorted_candidates = sort(candidates)
    return sorted_candidates
  else:                     // webpage new visit
    subdomain_node = get_subdomain_node(url)
    if subdomain_node != NULL:
      candidates = subres_nodes_of_the_subdomain
    else:
      candidates = subres_nodes_of_the_website
    sorted_candidates = sort(candidates)
    num_predicted = avg_num_of_webpage_children
    return sorted_candidates[0:num_predicted]
```

Figure 14: Pseudo Code of Subresource Prediction

predicted subresources, i.e., the subresource nodes that are shared by more webpage nodes). JavaScript and CSS files have higher priority than images because they may further request subresources and scripting may block later executions. Long URLs has higher chance to contain session dependent strings, which makes the URL useless next time. So long URLs have low priority.

To reduce the unnecessary data usage, speculative loader loads the predicted subresources adaptively. When the number of predicted resources is more than the number of allowed concurrent TCP connections, the resources with high priority will be requested immediately and other resources will be put into a waiting queue. If the main resource file is downloaded and parsed before the waiting resources are actually requested, the waiting queue will be updated with the actually needed resources, reducing unnecessary data usage from prediction misses.

6.3 Update Service

Update service constructs and modifies resource graphs in a metadata repository. Two major operations are performed on the nodes in the resource graph: *update* and *trim*. *Update* operation adds a node if the node does not exist in the resource graph or updates the information stored in the node if the node exists in the resource graph already. *Trim* operation removes the nodes that are not visited for more than one month from the resource graph.

After a webpage is open, update service *updates* the webpage nodes, its subresource nodes, the corresponding subdomain node and website node in the resource graph. Some webpages dynamically request subresources after they are open, e.g., by using AJAX. Update service can also capture those requests and update the subresource nodes accordingly. Every day, update service *trims* resource graphs and removes the stale nodes, whose last visited time is older than a month. *Trimming* resource graphs keeps the user viewed website resource graph up-to-date and limits the size of a metadata repository.

6.4 Temporary Cache

The purpose of temporary cache is to store the resources that have “no-store” in their cache-control header fields temporarily. Those files should not be stored in the cache. When speculative loader loads predicted resources, resources with “no-store” in their cache-control header fields will be saved to temporary cache and other resources will be saved to the normal cache. Later when the browser engine actually requests the speculatively loaded resources, it will get them either from the normal cache or temporary cache. After the webpage is open, all the resources in temporary cache will be deleted.

Chapter 7 **Evaluation**

We evaluate the Tempo design through a trace-based simulation, lab experiments and a field trial. The evaluation shows that the subresource prediction has both high accuracy and high coverage, resulting in 1 second (~20%) reduction of browser delay with negligible overhead.

7.1 Subresource Prediction Performance

We first evaluate the performance of the subresource prediction and the training period required by Tempo to make good predictions, based on the LiveLab traces. We employ two metrics: *hit ratio* and *usefulness*. As mentioned in Chapter 5.2, *hit ratio* represents prediction accuracy and *usefulness* represents the prediction coverage.

Figure 15 shows the weekly and monthly hit ratio and usefulness of subresource prediction, respectively. The first week's hit ratio (50%) and usefulness (56%) are already much higher than web prefetching shown in Chapter 5.2. The highest hit ratio (65%) and usefulness (73%) are reached in the third month. Interestingly, they drop slightly at week 4, 5, 12 and month 4, 5, largely when the LiveLab users visit a different set of websites around holidays and school breaks. Tempo takes time to construct the resource graph for the new websites.

7.2 Lab Experiments

We now evaluate how the subresource prediction performance is translated into the browser delay reduction through lab-based experiments. For the experiments, we port

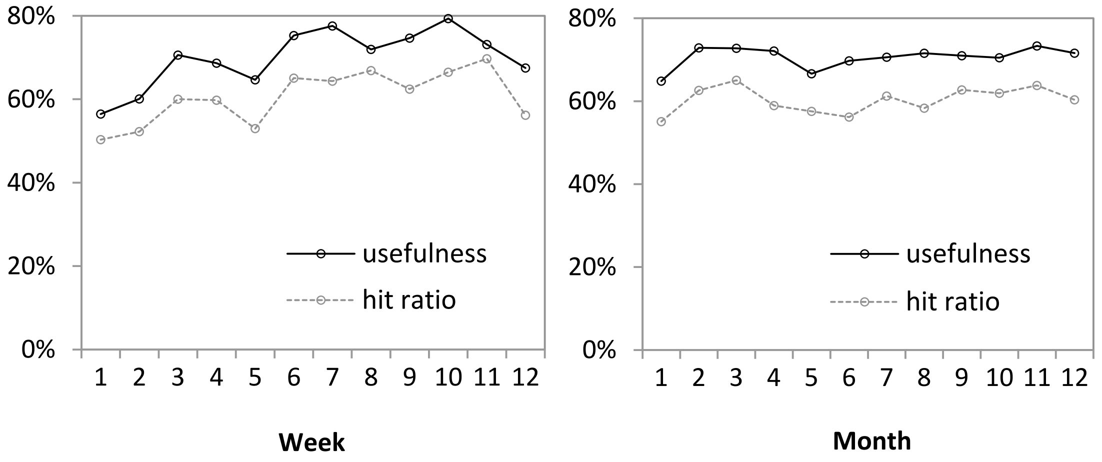


Figure 15: Hit ratio and usefulness of subresource prediction for the first 12 weeks (Left) and the entire year (Right). Each data point is the average value across 24 LiveLab users

Tempo to the Samsung Galaxy S II smartphone that runs Android Gingerbread [9]. The code of Tempo is instrumented to record the browser delay efficiently. All experiments use 3G network provided by AT&T, a major U.S. carrier, at our lab on Rice campus where 3G signal strength is strong.

7.2.1 Revisits

First, we show that Tempo can reduce the browser delay of webpage revisits to very close to the upper bound presented in Chapter 5.3. We use the homepages of the websites from Table 1 in the experiment. We first open the URLs in the browser once to warm up the cache and let the browser construct the resource graph. Then we open the URLs one by one for five times and calculate the average browser delay. Even though all the webpage visits are revisits in the experiment and we have minimized the time interval between revisits, there can still be cache misses due to the dynamic and/or session dependent content in the web. Subresource prediction cannot predict all the subresources needed, either. On average, the hit ratio is 65% and the usefulness is 72%, which are close to the prediction accuracy and coverage evaluated in Chapter 7.1.

We compare the browser delays between legacy loading and Tempo with three different cache states, similar to what were used in Chapter 5.3, i.e., fresh, expired, and empty. The browser is modified to always revalidate the resources stored in an expired cache and clears the cache before each webpage visit in an empty cache.

Table 2 shows the browser delays of webpage revisits under different cache states without speculative loading (*Legacy*) and with Tempo. With a fresh cache, the browser delays of Legacy and Tempo are close because most of the subresources are available locally. With an expired cache, Tempo reduces 25% (1445 ms) of browser delay on average. Tempo also reduces 24% (1464 ms) of the browser delay under an empty cache. Since 70% of the requested resources of a webpage from top 10 visited websites are either expired or not in the cache, as mentioned in Chapter 5.1, we estimate that Tempo can reduce the browser delay by around 1 second or ~20% with a realistic cache. This one-second reduction of the browser delay is also confirmed by our field trial, which will be discussed in Chapter 7.3.

Table 2: Browser delay reduction from Tempo for *webpage revisits* under different cache states (in ms)

	Sites	Legacy	Tempo	Reduction	
Empty Cache	ESPN	7031	5322	1709	24%
	CNN	6346	5307	1039	16%
	Google	3932	3257	675	17%
	Yahoo! Mail	5083	3442	1641	32%
	Weather	7167	4716	2451	34%
	Craigslist	3677	2470	1207	33%
	Neopets Games	10660	10220	440	4%
	Varsity Tutors	9987	7914	2073	21%
	Ride METRO	6945	5488	1457	21%
	Rice Registrar	6027	4084	1943	32%
	Average	**6686**	**5222**	**1464**	**24%**

	Sites	Legacy	Tempo	Reduction	
Expired Cache	ESPN	6748	5372	1376	20%
	CNN	5992	4274	1718	29%
	Google	3411	3073	338	10%
	Yahoo! Mail	5083	3265	1818	36%
	Weather	6109	3835	2274	37%
	Craigslist	3648	2089	1559	43%
	Neopets Games	10639	9280	1359	13%
	Varsity Tutors	8516	6677	1839	22%
	Ride METRO	6109	4620	1489	24%
	Rice Registrar	4169	3489	680	16%
	Average	**6042**	**4597**	**1445**	**25%**

	Sites	Legacy	Tempo	Reduction	
Fresh Cache	ESPN	3491	3602	-111	-3%
	CNN	4873	4507	366	8%
	Google	2407	2842	-435	-18%
	Yahoo! Mail	3239	3472	-233	-7%
	Weather	5055	4559	496	10%
	Craigslist	3123	2400	723	23%
	Neopets Games	9041	9076	-35	0%
	Varsity Tutors	5969	5384	585	10%
	Ride METRO	4220	3801	419	10%
	Rice Registrar	3046	3609	-563	-18%
	Average	**4446**	**4325**	**121**	**1%**

The browser delay reduction for each website mainly comes from the time waiting for the main resource to discover the subresources. Thus the content richness of the webpage, measured by the number of subresources and the webpage type, i.e., mobile or non-mobile, does not influence the browser delay reduction directly: most of the reductions in Table 2 are close to each other, i.e., ~1.4 seconds. The time spent to download and parse the main resource affects the discovery time of subresources. There are two main limiting factors. (*i*) The main resource redirection delays main resource download, e.g., the Weather website. (*ii*) The execution of JavaScript codes delays main resource parsing, e.g., the Varsity Tutors website. Since Tempo eliminates the resource dependencies, it can provide more browser delay reduction for websites that have previous two limiting factors.

The browser delay reduction of Tempo is very close to the upper bound presented in Chapter 5.3. Under an expired or an empty cache, Tempo can reduce the browser delay by around 1.4 seconds, which is 70% of the upper bound we can get, i.e., 2 seconds. For a realistic cache, Tempo can reduce the browser delay by around 1 second, which is 71% of the upper bound with realistic cache, i.e., around 1.4 seconds. By achieving its design goal, Tempo essentially keeps most subresources fresh in the cache when the browser requests them. Table 2 shows that the average browser delay of Tempo under an expired and an empty cache (4597 ms and 5222 ms) is only 3% and 17% larger than that of Legacy under a fresh cache (4446 ms), which is the ideal case. Tempo does overcome the limitation of caching.

Table 3: Browser delay reduction from Tempo for *new webpage visits* under different cache states (in ms)

Empty Cache	**Sites**	**Legacy**	**Tempo**	**Reduction**	
	ESPN	6162	4205	1957	32%
	CNN	7091	6054	1037	15%
	Google	4638	2945	1693	37%
	Yahoo! Mail	5572	5047	525	9%
	Weather	5357	5244	113	2%
	Craigslist	5163	3463	1700	33%
	Neopets Games	6914	6623	291	4%
	Varsity Tutors	14921	12674	2247	15%
	Ride METRO	6829	6171	658	10%
	Rice Registrar	6506	5534	972	15%
	Average	**6915**	**5796**	**1119**	**17%**

Expired Cache	**Sites**	**Legacy**	**Tempo**	**Reduction**	
	ESPN	3163	2788	375	12%
	CNN	3519	2438	1081	31%
	Google	2376	2492	-116	-5%
	Yahoo! Mail	4472	3162	1310	29%
	Weather	3757	2682	1075	29%
	Craigslist	2624	1848	776	30%
	Neopets Games	7326	7038	288	4%
	Varsity Tutors	10598	7437	3161	30%
	Ride METRO	3352	2602	750	22%
	Rice Registrar	4570	3672	898	20%
	Average	**4576**	**3616**	**960**	**20%**

Fresh Cache	**Sites**	**Legacy**	**Tempo**	**Reduction**	
	ESPN	3152	2587	565	18%
	CNN	2994	3328	-334	-11%
	Google	2982	2295	687	23%
	Yahoo! Mail	5222	5282	-60	-1%
	Weather	5180	3763	1417	27%
	Craigslist	1203	1210	-7	-1%
	Neopets Games	10105	9795	310	3%
	Varsity Tutors	7126	8013	-887	-12%
	Ride METRO	2759	3460	-701	-25%
	Rice Registrar	3929	3708	221	6%
	Average	**4465**	**4344**	**121**	**3%**

7.2.2 New Visits

Tempo can also greatly reduce the browser delay for new webpage visits, which accounts for 75% of the total webpage visits in LiveLab traces. In the experiment, we use the websites in Table 1. We first open the homepage of the website in the browser once to warm up the cache and let the browser construct the resource graph. Then we navigate to five other webpages in the same website and calculate the average browser delay. The browser delay for the homepage is not counted. Even though new webpages are used, and thus the subresources needed by the webpage will be different, subresource prediction can still predict some of the subresources needed from the shared subresource nodes. On average, the hit ratio is 50% and the usefulness is 67%, which is only slightly lower than the prediction accuracy and coverage evaluated in Chapter 7.1.

With similar cache states, we compare the browser delays between Legacy and Tempo. Table 3 shows the browser delays of new visits to webpages of different websites. Under a fresh cache, Legacy and Tempo exhibit similar browser delay. Under an expired and an empty cache, on average, Tempo incurs 20% (960 ms) and 17% (1119 ms) less browser delay than the Legacy, respectively. Notice that the browser delay of Tempo under an expired cache (3616 ms) is even 19% smaller than that of Legacy under a fresh cache (4465 ms). The reasons are that Tempo can effectively revalidate the expired subresources, warm up the TCP connections and thus download new subresources much faster.

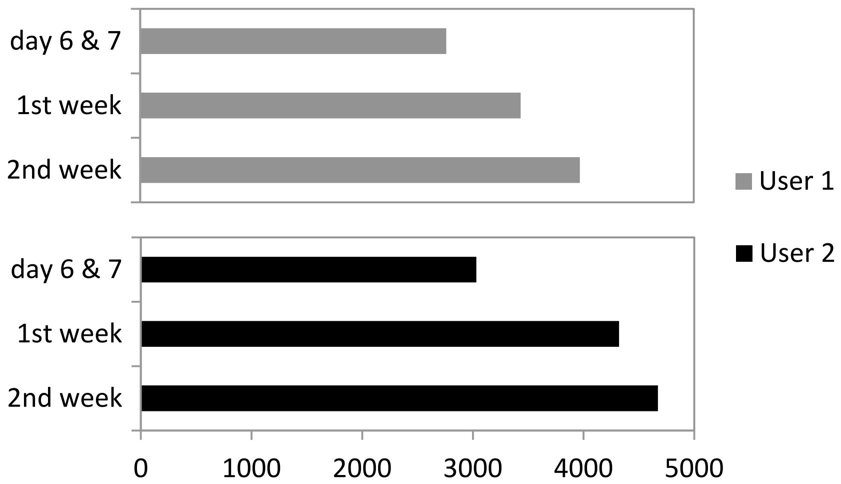


Figure 16: Average browser delays (ms) in different periods of the field trial. Speculative loading is enabled in the 1st week and disabled in the 2nd week

7.3 Field Trial

We also conduct a field trial to study the performance of Tempo browser. In the field trial, two Samsung Galaxy S II smartphones are used by two participants. Both smartphones run Android Gingerbread [9] with Tempo browser and use the 3G network provided by the U.S. wireless carrier AT&T. The field trial took two weeks for each participant. Speculative loading was enabled in the first week, but disabled in the second. The cache was cleared before the field trial and was never cleared during the field trial.

The results are shown in Figure 16. The average browser delay of the 2nd week is 433 ms longer than that of the 1st week and 1424 ms longer than that of day 6 & 7, the last two days in the 1st week. The results show that once Tempo has constructed the resource graphs, it outperforms Legacy by over one second, which is consistent with our findings from the lab experiments described in Chapter 7.2. The results also indicate that Tempo is effective with just several days' training period.

7.4 Overhead

Tempo incurs very low overhead of three types: performance, wireless data usage, and storage usage. Tempo incurs performance overhead when a predicted subresource is not actually needed. In this case, loading the predicted subresource will occupy a TCP connection, making subresources that are actually requested wait for available connections. We have minimized this overhead by prioritizing the predicted subresources and loading them adaptively, as discussed in Chapter 6.2. From the experiments presented in Chapter 7.2 and 7.3, it is also clear that the benefit from Tempo outweighs the overhead.

Tempo incurs data usage overhead when a predicted subresource is not actually requested. A higher hit ratio leads to lower data usage overhead, as discussed in Chapter 5.2. Resource prediction in Tempo achieves a hit ratio as high as 65%, as presented in Chapter 7.1, which is four times as much as that of web prefetching (16%). Though 35% of the predicted subresources are not actually needed by the current webpage, the data usage overhead is usually even lower because of three reasons: (*i*) the predicted subresources are loaded adaptively, which minimizes the data usage overhead; (*ii*) the predicted subresources are widely shared by different webpages of the same website, effectively amortizing the overhead over multiple webpages; (*iii*) the predicted subresources are visited before and they are likely to be still in the cache, resulting in little network traffic for expired resources or even none for fresh resources. We find the data usage overhead in the field trial to be as low as 0.7 MB per week.

Tempo incurs storage usage overhead by constructing and storing metadata repository on the mobile device, which requires additional flash storage space. However, the additional storage is small because metadata repository does not contain actual resource content, as discussed in Chapter 6.1. For each of the 24 LiveLab users, one year's metadata repository takes only 165 KB on average (max 576 KB), which is negligible in view of what is available to modern mobile devices.

Chapter 8 **Related Work**

8.1 Browser Speed Characterization

The performance of browsers has attracted a lot of interest from both industry and academia. Existing work, however, is limited in both scope and method. Characterization work on PC browsers assumes resource loading is negligible for PCs with enterprise Ethernet and therefore focuses on the compute-intensive IR operations [22, 30, 48, 59]. The Internet Explorer (IE) team [48] focuses on the computation of the browser and provides a breakdown of the CPU cycles consumed by the key IE subsystem. The network improvement is discussed separately and is not clearly included in the breakdown. Using call stack sampling for performance characterization, the authors of [22, 30] throw out the network time in their analysis since their profiling method cannot capture the time spent idling. None of their methods [22, 30, 48, 59] capture the cost of resource loading or consider the concurrency of operation execution as discussed in Chapter 2.1. In contrast, we 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 mobile devices.

Huang et al. [20] investigate smartphone browser performance mainly from the network perspective without looking into the internals of the browser. They quantify how the browser performance is affected by the network RTT, the packet loss rate, the number of concurrent TCP connection, and the resource content compression, without an understanding of how the browser operates and interacts with the OS services including the network stack on mobile devices. The authors, however, measure 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 misses the latency of the browser initialization for the webpage before the first DNS packet is sent out, i.e., typically around 200ms on G1, and the latency of all operation executions after the reception of the last packet, which can take up to 2 seconds according to our measurement. Moreover, the authors approximate computing time by the browser with the TCP idle time, or periods having no network activity. This approximation will miss a significantly portion of the computing time by the browser, i.e., 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.

8.2 Browser Speed Improvement

8.2.1 Solutions with Infrastructure Support

Many have studied ways to improve browser speed, in particular resource loading. While only very few have specifically targeted mobile browsers, we discuss related work in terms of their approaches. Most proposals require infrastructure support, either from the web server or a proxy, e.g., thin-client approaches [24, 27, 38, 46] and session-level techniques [43]. Web prefetching with infrastructure support is also widely studied, e.g. [2, 6, 11, 29, 39], and is used in real world [13, 15, 35, 52]. Amazon Kindle Silk [2] has a split browser architecture, which can be accelerated by utilizing the infrastructure support on the server side, including web prefetching. However, the review [47] shows that Silk is actually faster when the acceleration is turned off, i.e., without infrastructure support. The review also shows that Silk is not faster than the browser on other tablets, e.g., the browser on iPad or Galaxy Tab. In a spirit similar to prefetching, *Crom* [32] speculatively runs JavaScript event handlers, prefetches the web data and pre-uploads local files, also with server help. TCP fast open [41] modifies the TCP protocol to decrease the network latency by one full RTT. A recent protocol proposal, *SPDY* [51], improves the web performance by providing multiplexed streams, request prioritization, HTTP header compression, server push and server hint. It does so by adding a session layer atop of SSL and requires changes on both client and server. Though the approaches discussed above are effective, they are hard to deploy, are subject to the ability of the servers, cannot provide end-to-end security or has limited client JavaScript support.

8.2.2 Client-Only Solutions

Client-only solutions are attractive because they can be immediately deployed and work with existing web content. The authors of [30, 48, 59] seek to improve the client speed of compute-intensive operations in browsers. As we showed in Chapter 3.3, their solutions will lead to negligible improvement in mobile browser speed.

There are two orthogonal types of client-only solutions to improve resource loading: hardware improvement and RTT hiding. Hardware improvement speeds up resource loading by providing faster network stack and other OS services. The improvement is likely to continue because of Moore’s Law. A browser needs to access the mass storage on a mobile device when it accesses the data in the cache or cookie. But the performance of the mass storage in the mobile device is already good enough and has little impact on the browser delay. The authors of [23] show that the mobile browser delay over WiFi can vary up to 500% by varying the flash storage used on the smartphone, e.g., run the Android system on an external SD card instead of the internal storage. However, their finding only indicates that a bad flash storage has significant negative impact on browser performance. In general, flash storage systems are already good enough and their improvement will not bring any benefit to browsers because the flash storage system on mobile devices incurs nearly no CPU IOwait time, which is the main contributor to the long browser delay with a bad storage system.

This thesis focuses on the client-only solutions for RTT hiding to speed up resource loading without any hardware change on mobile devices. Existing client-only solutions targeted at RTT hiding employ one or both of the following two approaches. *Browser caching* [40] is the most widely used client approach. As we show in Chapter 5.1, caching is not effective for mobile browsers because of the long RTT and the large percentage of revalidations [55]. *Web prefetching* can also be implemented without server support. However, as we show in Chapter 5.2 and also observed by others on PCs [33], client-only prefetching introduces large data usage overhead with little performance improvement because of the low prediction accuracy.

8.3 Other Browser Research

Many have studied browsers beyond performance characterization and improvement. We next discuss three other important research directions involving browsers: browser resource management, browser security, and browser extensions and plug-ins.

Browser resource management becomes important as the browser evolves to a multi-principal operating environment, where multiple principals (websites) interact programmatically in a single webpage. The principals share the browser's resources, such as network, display, memory, and CPU, on the device. MashupOS [53] employs new abstractions to secure the browser resource sharing. ServiceOS [34] proposes DOM-recursive resource allocation policies to manage a browser's resources.

As a browser has become a platform for applications, old browser designs expose many security vulnerabilities. Many researchers have redesigned the browser architecture to address the security problems. Their solutions usually isolate web applications in multiple OS processes, making the browser more robust. The un-trusted parts can run in a low-privilege sandbox, limiting the damages and the exposure of browser vulnerabilities. The secure web browser in SubOS [21] is modularized and it utilizes the SubOS processes to isolate different modules. Tahoma [8] introduces a trusted browser operating system (OS) layer on which browser implementations can run and the browser instances are isolated by virtual machine sandboxing. OP browser [18] leverages OS design principles and partitions the browser into smaller subsystems. The interactions of each subsystem are limited by OS-level sandboxing techniques. Chromium [5] modularizes the browser into two protection domains and protects the user domain by sandboxing the web domain.

Later, Chromium [42] further introduces the abstractions of web programs and isolates them from each other using process. Chromium OS [17] not only leverages the security features in Chromium but also undertakes a wide-range of other security features at all levels, e.g., OS hardening and verified boot. Gazelle [54] constructs the web browser as a multi-principal OS and separates each website principal into discrete protection domains. IBOS [49] provides an OS and browser co-design, which pushes browser-level abstractions down to the lowest software layer in the OS, providing a clean separation between browser functionality and browser security.

In addition to secure browser architecture design, there are also other research works tackling browser security problems. For example, Object views [31] enable the secure sharing of fine-grained JavaScript objects among different principals, i.e., websites, in the same webpage. WebAnalyzer [45] crawls and inspects webpages to find out the unsafe features caused by the incoherent browser access control policies.

Browser extensions and plug-ins are very popular; they extend the functionality of a browser. However, running legacy or native codes as browser extensions and plug-ins is a big challenge. Xax [10] employs OS services and a browser plug-in to run legacy codes in highly restricted processes. Native Client [58] runs un-trusted x86 native codes as browser extensions in a sandbox, delivering native code performance for browsers securely. The Plug-In Development Kit of HP webOS [19] allows developers to convert C/C++ codes to browser plug-ins that work with JavaScript-based applications.

Browser extensions and plug-ins are usually vulnerable to attacks. The authors of [4] propose a new extension system to protect the benign extensions from attackers by providing least privilege, privilege separation and strong isolation features. VEX [3] utilizes static information-flow analysis to find vulnerabilities in browser extensions.

Chapter 9 **Discussion**

Speculative loading leverages concurrent TCP connections available in the browser to fully parallelize resource loading. The performance benefit of speculative loading will increase as more concurrent TCP connections are available to mobile systems because more subresources can be speculatively loaded or revalidated. With four current TCP connections, Android Gingerbread's default browser has the fewest concurrent TCP connections [7] compared to other popular mobile browsers. So Tempo is likely to provide other mobile browsers with more performance improvement than Android Gingerbread's default browser reported in our work.

Though we demonstrate the browser delay reduction from Tempo using current 3G networks, Tempo will benefit mobile browsers using faster networks with a higher bandwidth and a shorter RTT, e.g., WiFi and 4G networks. Higher bandwidth can support mobile browsers with more concurrent TCP connections, which will make speculative loading more powerful. Shorter RTTs make the subresource speculation process much faster, leading to even shorter browser delay and larger browser delay reduction.

Tempo aims at reducing the delay of opening a webpage. After the webpage is opened, the interaction between the user and the webpage is also very important, especially for web applications. Web applications leverage asynchronous network requests and JavaScript to provide users with a nice interactive experience. However, instant response from a web application is still a big challenge. Smart speculation and caching are needed to hide long RTTs for network requests. Better designs of JavaScript and browser engines are necessary to provide smooth navigation inside web applications.

Chapter 10 **Conclusion**

We characterize the performance of mobile browsers and find that the bottleneck is resource loading instead of compute intensive operations. Of solutions for browser speed improvement, client-only ones are immediately deployable, scalable, and secure. It has been well known that client-only solutions are not as effective in improving speed as ones with infrastructure support. Leveraging an unprecedented data set of mobile web usage, we provide the first comprehensive treatment of the effectiveness of client-only solutions.

We demonstrate the ineffectiveness of browser caching and client-only web prefetching on mobile browsers. Caching is not effective because many of the resources are not in the cache or their cached copies quickly expire. Client-only web prefetching is harmful because it results in significant additional wireless data usage with little performance improvement.

In order to address the limitations of the previous two approaches, we propose speculative loading, a client-only approach that predicts the subresources of a webpage given its URL is provided and then loads the subresources simultaneously with the main HTML. Our implementation of speculative loading, Tempo, can reduce browser delay by 1 second (~20%) under 3G networks.

Finally, we empirically show that the upper bound of browser delay reduction for client-only solutions is 1.4 seconds. Our result suggests that it is imperative to involve the infrastructure in further improving mobile browser performance.

REFERENCE

[1] "Hypertext Transfer Protocol -- HTTP/1.1," http://www.ietf.org/rfc/rfc2616.txt.

[2] Amazon Silk: http://amazonsilk.wordpress.com/.

[3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX Conference on Security*, 2010.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[5] A. Barth, C. Jackson, C. Reis, and Google Chrome Team, "The Security Architecture of the Chromium Browser," 2008.

[6] C. Bouras, A. Konidaris, and D. Kostoulas, "Predictive Prefetching on the Web and Its Potential Impact in the Wide Area," *World Wide Web,* vol. 7, pp. 143-179, 2004.

[7] Browserscope: http://www.browserscope.org/.

[8] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy, "A safety-oriented platform for Web applications," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[9] CyanogenMod Wiki, "Samsung Galaxy S II," http://wiki.cyanogenmod.com/wiki/Samsung_Galaxy_S_II.

[10] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[11] L. Fan, P. Cao, W. Lin, and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: potential and performance," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999.

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

[13] Google, "Announcing Instant Pages," http://googlewebmastercentral.blogspot.com/2011/06/announcing-instant-pages.html.

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

[15] Google, "Web Developer's Guide to Prerendering in Chrome ": http://code.google.com/chrome/whitepapers/prerender.html.

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

[17] Google Inc., "Chromium OS," https://sites.google.com/a/chromium.org/dev/chromium-os.

[18] C. Grier, S. Tang, and S. T. King, "Secure Web Browsing with the OP Web Browser," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[19] Hewlett-Packard Company, "HP webOS," https://developer.palm.com/.

[20] 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.

[21] S. Ioannidis and S. M. Bellovin, "Building a Secure Web Browser," in *Proceedings of the FREENIX Track of 2001 USENIX Annual Technical Conference*, 2001.

[22] 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.

[23] H. Kim, N. Agrawal, and C. Ungureanu, "Examining storage performance on mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld)*, 2011.

[24] J. Kim, R. A. Baratto, and J. Nieh, "pTHINC: a thin-client architecture for mobile wireless web," in *Proceedings of the 15th international conference on World Wide Web*, 2006.

[25] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007.

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

[27] A. M. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya, "Improving web browsing performance on wireless pdas using thin-client compu-

ting," in *Proceedings of the 13th international conference on World Wide Web*, 2004.

[28] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "WebProphet: automating performance prediction for web services," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*.

[29] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "PocketWeb: instant web browsing for mobile devices," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[30] 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.

[31] L. A. Meyerovich, A. P. Felt, and M. S. Miller, "Object views: fine-grained sharing in browsers," in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[32] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster web browsing using speculative execution," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[33] J. C. Mogul, "Hinted caching in the web," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, 1996.

[34] A. Moshchuk and H. J. Wang, "Resource Management for Web Applications in ServiceOS," *Technical Report,* 2010.

[35] Mozilla, "Link prefetching FAQ," https://developer.mozilla.org/en/Link_prefetching_FAQ.

[36] 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.

[37] NYTimes, "Amazon Cloud Failure Takes Down Web Sites," http://bits.blogs.nytimes.com/2011/04/21/amazon-cloud-failure-takes-down-web-sites/.

[38] Opera Mini: http://www.operamini.com/.

[39] 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.

[40] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*, 2003.

[41] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP Fast Open," in *Proceedings of the ACM CoNEXT Conference*, 2011.

[42] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[43] P. Rodriguez, S. Mukherjee, and S. Ramgarajan, "Session level techniques for improving web browsing performance on wireless links," in *Proceedings of the 13th international conference on World Wide Web*, 2004.

[44] 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.

[45] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the Incoherencies in Web Browser Access Control Policies," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

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

[47] S. Souders, "Silk, iPad, Galaxy comparison," http://www.stevesouders.com/blog/2011/12/01/silk-ipad-galaxy-comparison/.

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

[49] S. Tang, H. Mai, and S. T. King, "Trust and protection in the Illinois browser operating system," in *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, 2010.

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

[51] The Chromium Projects, "SPDY: An experimental protocol for a faster web," http://www.chromium.org/spdy.

[52] W3C, "HTML5 Link type prefetch," http://www.w3.org/TR/html5/links.html#link-type-prefetch.

[53] H. J. Wang, X. Fan, J. Howell, and C. Jackson, "Protection and communication abstractions for web browsers in MashupOS," in *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, 2007.

[54] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The Multi-principal OS Construction of the Gazelle Web Browser," in *Proceedings of the 18th conference on USENIX security symposium*, 2009.

[55] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, "How effective is mobile browser cache?," in *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students, for the students (S3)*, 2011.

[56] Z. Wang, X. Lin, L. Zhong, and M. Chishtie, "Why are web browsers slow on smartphones?," in *Proceedings ACM Int. Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

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

[58] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[59] 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.