HTTP/2 Prioritization and its Impact on Web Performance

Maarten Wijnants
Hasselt University – tUL
Expertise Centre for Digital Media
Diepenbeek, Belgium
maarten.wijnants@uhasselt.be

Robin Marx
Hasselt University – tUL
Expertise Centre for Digital Media
Diepenbeek, Belgium
robin.marx@uhasselt.be

Peter Quax
Hasselt University – tUL – Flanders Make
Expertise Centre for Digital Media
Diepenbeek, Belgium
peter.quax@uhasselt.be

Wim Lamotte
Hasselt University – tUL
Expertise Centre for Digital Media
Diepenbeek, Belgium
wim.lamotte@uhasselt.be

ABSTRACT

Web performance is a hot topic, as many studies have shown a strong correlation between slow webpages and loss of revenue due to user dissatisfaction. Front and center in Page Load Time (PLT) optimization is the order in which resources are downloaded and processed. The new HTTP/2 specification includes dedicated resource prioritization provisions, to be used in tandem with resource multiplexing over a single, well-filled TCP connection. However, little is yet known about its application by browsers and its impact on page load performance.

This article details an extensive survey of modern User Agent implementations, with the conclusion that the major vendors all approach HTTP/2 prioritization in widely different ways, from naive (Safari, IE, Edge) to complex (Chrome, Firefox). We investigate the performance effect of these discrepancies with a full-factorial experimental evaluation involving eight prioritization algorithms, two off-the-shelf User Agents, 40 realistic webpages, and five heterogeneous (emulated) network conditions. We find that in general the complex approaches yield the best results, while naive schemes can lead to over 25% slower median visual load times. Also, prioritization is found to matter most for heavy-weight pages. Finally, it is ascertained that achieving PLT optimizations via generic server-side HTTP/2 re-prioritization schemes is a non-trivial task and that their performance is influenced by the implementation intricacies of individual browsers.

CCS CONCEPTS

• **Information systems → Browsers**; • **Networks → Application layer protocols**; **Packet scheduling**; **Network performance evaluation**; *Traffic engineering algorithms*; *Public Internet*; • **Computer systems organization** → Client-server architectures;

KEYWORDS

HTTP/2; Web Performance Optimization (WPO); Page Load Time (PLT); resource loading; prioritization; experimental evaluation

ACM Reference Format:
Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. 2018. HTTP/2 Prioritization and its Impact on Web Performance. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186181

1 INTRODUCTION

It has long been known that good webpage load performance is paramount to a satisfactory User Experience (UX) [20, 28, 34, 40, 43]. A hallmark of the webpage load process is that not all of the composing resources have equal utility and purpose, as some need to be processed and executed at client side (e.g., HTML, CSS, JavaScript (JS)) while others chiefly contribute to the visual completeness of the page (e.g., images, fonts). Since not all resources can be downloaded at the same time and the average webpage size as well as the resource count per individual webpage continues to grow [25], the correct prioritization of resource delivery is an important research topic.

When the time came to design a replacement for the aging HTTP/1.1, the creators of SPDY [41] and its successor, HTTP/2 [24], recognized the significance of prioritization. Next to other Web performance-related changes, such as switching to a binary format and the new Server Push mechanism [35], HTTP/2 aims to make optimal use of a single TCP connection (as opposed to the 6-17 parallel connections typically seen for HTTP/1.1 [8]) by allowing resources to be multiplexed and their data interleaved. To mitigate contention at the TCP layer among multiplexed assets, the protocol encompasses a prioritization system (see Section 3) that empowers the client to specify resource delivery precedence. HTTP/2's prioritization paradigm hence holds great potential as a viable instrument to optimize the Page Load time (PLT) of webpages. However, despite this potential, HTTP/2 prioritization has to date received little attention from the academic community. Even very recent scholarly work on resource (re-)prioritization [27, 31, 36] uses non-standard methods or complex workarounds with various inherent downsides (see Section 2), instead of relying on HTTP/2's built-in mechanism.

To this end, this paper addresses two research questions. The first one revolves around contemporary User Agents and how they exploit the HTTP/2 prioritization affordances when loading webpages (see Section 4). We found that there are roughly three extremely disparate methods in use by the most popular browsers today. This diversity is interpreted to be an indication that the optimal prioritization approach might be dependent on the browser's internal implementation. In effect, loading a webpage involves an interplay between a multitude of correlated processes, only one of which is network delivery [51, 52]. The second research question then focuses on investigating the impact of a total of eight prioritization algorithms on PLT performance. The investigated algorithms encompass both real-world implementations (e.g., the default HTTP/2 prioritization schemes utilized by Chrome and Firefox) and custom implemented variations thereof, as well as more naive approaches. This research question is tackled by conducting a full-factorial experimental benchmark on a corpus of 40 pages, tested using Chrome and Firefox under heterogeneous network conditions (see Sections 5 and 6).

2 BACKGROUND AND RELATED WORK

2.1 The Webpage Load Process

Modern webpages are often comprised of a large amount of heterogeneous resources (e.g., CSS stylesheets, JS code, fonts, images). Due to network limitations, these resources cannot all be downloaded concurrently. This leads to the need for User Agents to somehow prioritize the resource delivery order, as typically not all resources are needed to start rendering or even interacting with the page. From a user experience perspective, browsers might want to prioritize assets that have a direct visual impact on the webpage [9, 27, 43], such as images and fonts. However, these assets are often positioned using CSS stylesheets, which need to be processed first to provide an acceptable graphical layout to the user. Similarly, JS code might not have a large visual impact, but often adds functionality which the user needs to successfully accomplish her task in the webpage. Since CSS and JS resources can alter the webpage content, User Agents are forced to be conservative by blocking HTML parsing and/or the rendering of the page until those types of resources have been processed [51].

To aggravate matters, the browser does not know the full list of a webpage's constituent resources upfront; instead, it incrementally discovers them by evaluating previously received resources. In particular, parsing the HTML code of a webpage will reveal references to embedded resources, some of which can include resources on their own (e.g., import statements or font declarations in CSS files, JS modules, XHR fetches [42]).

2.2 Resource Prioritization Schemes

2.2.1 User Agents. Web browsers harness a complex set of heuristics to prioritize requests. These heuristics have grown as best practices for the HTTP/1.x protocols, which suffer from Head-Of-Line (HOL) blocking. This means a resource needs to be fully downloaded before another file can be fetched over the connection and prevents dynamic re-prioritization of in-flight resources. Because HTTP/1.x lacks dedicated prioritization features, resources are implicitly prioritized by requesting them in the desired order, often fragmented across multiple TCP connections. To derive this order, browsers traditionally use a speculative parser [3] to discover all referenced resources in the HTML code and then sort them based on the heuristics.

In contrast, HTTP/2 solves the HOL blocking problem by coalescing resources on a single underlying TCP connection. HTTP/2 includes an explicit resource prioritization mechanism based on dependency relationships and weights (see Section 3). Our study shows that modern User Agents map the HTTP/1.x-era heuristics and request orders onto HTTP/2 priorities in widely differing manners (see Section 4).

2.2.2 Experimental Improvements. While the browsers' heuristics are grounded in years of empiric expertise and therefore work well in the general case, they are suboptimal in many ways. For example, the implementations often do not take the dynamic factors of computation and network quality into account. Additionally, they are mostly coarse-grained, prioritizing resources based on their type instead of their individual impact on the page load. In response, a large body of prior work has sought to improve these default browser behaviors.

WProf instruments the browser to capture resource load dependencies and as such extracts the "critical path" for a page load. Resources on this critical path should be given maximal priority, as only their delivery can improve the observed performance [51]. Netravali et al. query the browser's JavaScript engine to capture more fine-grained resource dependencies, with the intent of reducing the time client-side computation needs to wait for the network [31]. Similarly, Ruamviboonsuk et al. prioritize the delivery of resources that incur a heavy processing cost, aiming to keep both the CPU and network fully utilized at all times [36]. So-called "split-browser" systems first actively rewrite and execute resources on the server before sending optimized resource bundles to the client [12, 32, 37, 38, 53]. Finally, several works focus on improving user perceived performance and employ either the concept of explicit user preferences (combined with fast visual load progress) [9] or explicitly track the user's focus of attention [27] to prioritize individual resources.

2.2.3 Priority Enforcement. The systems discussed in Section 2.2.2 typically include a proprietary implementation to enforce the resource priorities they calculate, which requires them to circumvent the browsers' existing heuristics and network behavior. Remarkably, despite the fact that most of these implementations employ HTTP/2 or SPDY, many of them neglect the protocol's built-in prioritization system, instead relying on non-standard methods or complex workarounds that can introduce new problems. For example, some implementations [22, 31, 36] use a JS-based client-side scheduler that fetches resources using `XmlHTTPRequests` (XHRs), bypassing the browser's built-in streaming engine and thus incurring additional delays. Others rely heavily on the new HTTP/2 Server Push feature [24], allowing them to partly solve both the resource discovery and prioritized delivery problems at the same time. However, Push suffers from complex interactions with the underlying TCP connection [6], potentially leading to bandwidth contention and hence impairing network performance [9, 27, 35, 36]. Finally, split-browsers often require a custom browser implementation [38, 53] or the installation of a local proxy [32, 37]. Only two papers mention HTTP/2 prioritization at all [9, 36], but indicate that they simply use the default browser implementation.

The limited adoption of the HTTP/2 prioritization system might be explained by the lack of easy-to-use interfaces to directly interact with it in a standards-complaint way, though such interfaces are being worked on [33, 56]. Existing Web standards (e.g., `preload`, `async`/`defer` JS, Service Workers) only allow indirect access or map suboptimally to HTTP/2 priorities in present-day implementations (see Section 4.4). The only fully standards-compliant method (without resorting to a custom User Agent implementation) is then to have the server override the client-issued priorities, which is perfectly acceptable behavior according to the HTTP/2 specification [24].

Contrary to existing academic efforts, our work does not aspire to draft elaborate resource dependency graphs, but rather uses a custom server implementation to directly manipulate the HTTP/2 prioritization directives emitted by off-the-shelf User Agents. As such, our work is complementary to the related work and offers insights into a more standards-compliant avenue for their implementation.

2.3 HTTP/2 Prioritization

While there exists a large body of research into the performance of HTTP/2 and its predecessor SPDY [11, 13, 18, 29, 35, 44, 52], only a few papers investigate their standard prioritization mechanisms. Wang et al. [52] conclude that the impact of explicit (re-)prioritization is minimal because the internal browser implementation inherently limits the order in which resources can be processed. Our work partly confirms this, but also shows that mis-prioritization can lead to severe performance penalties (see Section 6.1). Bergan [5] published the only in-depth benchmarking effort, comparing the PLT performance of HTTP/2 prioritization against a server-side lottery scheduler [50]. Only for 31% of the test corpus was HTTP/2 prioritization found to yield an improvement. Our work considers more realistic generic resource scheduling algorithms and factors heterogeneous network conditions and multiple User Agent implementations into the evaluation.

3 HTTP/2'S PRIORITIZATION SYSTEM

In HTTP/2, each resource (including its request and response) is conceptually carried over a unique *stream*. Resource content is transmitted in relatively small chunks to enable data pertaining to concurrent HTTP/2 streams to be delivered in an interleaved fashion over the unified TCP connection. Multiplexing data this way can however cause contention among in-flight resources to emerge. In response, the HTTP/2 specification includes a flexible resource prioritization model [24]. The client (e.g., User Agent) can draft a *dependency tree* to inform the resource origin (i.e., Web server) about the way it would prefer the delivery of concurrent streams to be handled. Each HTTP/2 stream is identified by a unique ID and added to the dependency tree, of which the root node (with ID 0) denotes the TCP connection. Nodes are added as new resources are requested and removed when their corresponding resources have been fully downloaded.

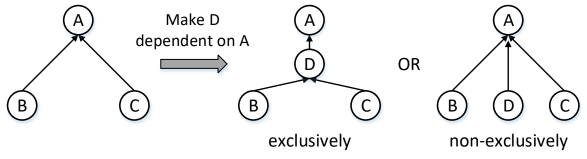


Figure 1: HTTP/2 dependencies: *exclusive* vs *non-exclusive*.

The first facet of HTTP/2's prioritization model is the establishment of dependencies among streams. The existence of a parent-child relationship among two resources indicates that the transmission of the dependent resource must be postponed until its ancestor has been downloaded completely (or until it is temporarily impossible to make progress on the parent). Alternatively, a sibling relationship between nodes indicates that the corresponding resources must be delivered in parallel. The dependency relationships can be marked to be either *exclusive* or *non-exclusive* by the client. An exclusive dependency between ancestor A and dependent D will render D the sole child of A, with A's original children (if any) becoming dependent on D (see Figure 1 and also the tree in Figure 2 (a), which was constructed using only exclusive dependencies). If instead a non-exclusive dependency would have been installed, D would have become a sibling of A's original children (see Figure 1 and also the dependency tree in Figure 2 (b)).

The second element of HTTP/2's prioritization approach, stream weighting, then allows fine-grained interleaved delivery of sibling resources. Siblings should be allocated network bandwidth proportionally to their weight, which can take on any natural number in the [1,256] range. For instance, given three siblings X, Y and Z with weight values 10, 20 and 30, the client would like these streams to respectively be granted one sixth, one third and half of the available bandwidth.

HTTP/2's prioritization model paves the way for two naive approaches. First, making each resource *exclusively* dependent on the previously requested one produces a serial dependency tree where each node has at most one parent and one child, yet never any siblings. This model, referred to as **First-Come First-Served (FCFS)**, causes resources to be downloaded integrally before continuing with the next one in the request chain. FCFS is the standard per-TCP-connection behavior of HTTP/1.x. Secondly, by only installing *non-exclusive* dependencies on the root, a broad tree is constructed where resources never have any children, only siblings. Combined with a uniform weighting scheme, this **Round Robin (RR)** model yields a completely fair distribution of the bandwidth budget among pending resources; this is in fact the default prioritization logic recommended by the HTTP/2 specification [24].

4 USER AGENT IMPLEMENTATIONS

User Agent vendors are free to exploit the prioritization mechanism included in the HTTP/2 standard in any way they find appropriate. In this section, we will look at how the most popular Desktop and Mobile browsers [4, 39] approach HTTP/2 prioritization. Our findings are summarized in Table 1 and were obtained via a combination of empirical analysis (i.e., HTTP/2 network traffic inspection) and source code review. With the exception of Opera, all multi-platform browsers behave equivalently in their Desktop and Mobile versions

Table 1: Browser prioritization strategies. Identical in Desktop and Mobile versions unless indicated.

Strategy	User Agents
Dynamic FCFS	Chrome 58, Opera 48 (Desktop), Brave 1 (Mobile), Dolphin 12 (Mobile)
Naive RR	Internet Explorer (IE) 11 (Desktop), Edge 40, Opera Mini 30 (Mobile)
Weighted RR	Safari 11, UC browser 11.4 (Mobile)
Tree-based	Firefox 54

Table 2: Top: Chrome 58's resource classification in priority buckets, with associated HTTP/2 weight values. Bottom: Resource type weighting in Safari 11 (left) and Firefox 54 (right).

Bucket	Content Types	Weight
`Lowest`	pushed assets (initially)	110
`Low`	images, `async` and `defer` JS	147
`Normal`	JS declared after first image	183
`High`	JS declared prior to first image, XHR	220
`Highest`	HTML, CSS, fonts	256

Weight	Content Types	Weight	Content Types
16	pushed assets	2	pushed assets (initially)
8	images	22	images
16	fonts, XHR	32	HTML, JS, CSS, XHR
24	JS, CSS	42	fonts
255	HTML		

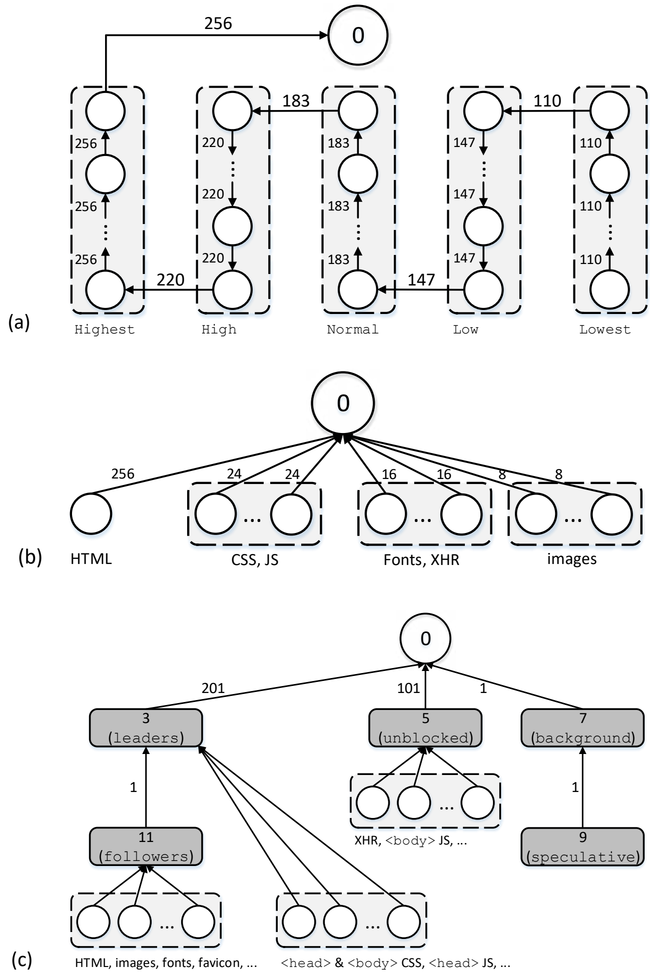


Figure 2: HTTP/2 dependency tree layout produced by (a) Chrome 58, (b) Safari 11 and (c) Firefox 54.

(Safari was tested on iOS, Edge on Windows Phone, the others on Android). As many specialty browsers (e.g., Opera and Brave) use (parts of) the source code of the more established projects, we will only zoom in on the leading implementations.

4.1 Chrome: Dynamic FCFS

Google Chrome generates completely linear HTTP/2 dependency trees, akin to the naive FCFS approach. Chrome categorizes webpage assets in five *priority buckets* (see Table 2) and resorts to *exclusive* dependencies on both an intra- and inter-priority bucket level (see Figure 2 (a)). As such, requests for webpage resources belonging to the same bucket are made exclusively subordinate to each other in request order, with the oldest still outstanding request in a priority bucket depending exclusively on the most recent pending request in a higher priority bucket . This results in a dynamic FCFS variant, where the transmission of in-flight lower-priority resources can be pre-empted by emerging resources in higher-priority buckets. The net intent of Chrome's HTTP/2 prioritization strategy is non-interleaved transmission, in strict order of decreasing resource priority. All resource requests that fall in a particular priority bucket will share the same HTTP/2 weight value (see Table 2). These weight values are inconsequential in practice, as nodes in the dependency tree will never have siblings.

4.2 IE, Edge and Safari : (Weighted) RR

IE, Edge and Safari take an almost orthogonal approach to that of Chrome. They generate wide, completely parallel dependency trees, using only *non-exclusive* relationships.

IE and Edge in fact do not include any prioritization information in their requests. Assuming a standards-compliant server implementation, a fallback to the default prioritization behavior as stipulated in the HTTP/2 specification (i.e., naive RR) will then be enacted: each resource will be made *non-exclusively* dependent on the root node with weight 16 [24].

Safari is more intelligent in that it does explicitly specify different weights for individual resource types, as shown in Table 2 and Figure 2 (b). It however lacks some of Chrome's nuances (e.g., not differentiating between JS resources declared before and after the first image in the webpage).

4.3 Firefox: A Class Apart

Firefox is unique in our survey, in that it is the only User Agent that builds a complex, multi-layered dependency tree. Upon HTTP/2 connection establishment, Firefox organizes the dependency tree in a predefined layout by opening a total of five perpetually idle streams. The resulting prioritized *phantom nodes* are then used as parents to cluster resource requests using *non-exclusive* dependency relationships. This clustering is based on a combination of resource type and declaration location in the encompassing HTML document, see Figure 2 (c). For example, the initiating HTML request is assigned to the `followers` class (represented by the idle HTTP/2 stream with ID 11), CSS assets (declared in either the `<head>` or `<body>` of the HTML document) and `<head>` JS files are grouped in the `leaders` class (idle stream ID 3), while XHR and `<body>` JS resources belong to the `unblocked` class (idle stream ID 5). To differentiate between heterogeneously typed siblings clustered under the same phantom node, Firefox employs a 5-level weighting strategy, which is non-exhaustively summarized in Table 2. As such, each individual phantom node employs a weighted RR prioritization setup and siblings under the same phantom parent are serviced in parallel, sharing bandwidth according to their weights.

4.4 Contemporary Web Technologies

The previous sections show that HTTP/2 prioritization is (semi-)well defined for basic webpage loading in Safari, Chrome and Firefox. However, prioritization support for cutting-edge Web techniques that hold the power to influence resource request order was empirically found to often still be lacking.

For example, HTTP/2 Server Push, which can be used to proactively transmit resources to the client before having received a request for them [6, 24] and which is popular in previous work [9, 27, 36], seemed to work as expected in Chrome but not in Firefox or Safari. While Chrome eventually promotes pushed resources to their rightful priority bucket (and hence place in the serial dependency tree) based on their type, Firefox makes them *non-exclusively* dependent on the root node (i.e., as a direct sibling of the top-level phantom nodes) with a weight that eventually switches to the correct content type-driven value (see Table 2). This can lead to pushed resources contending for bandwidth with other top-level nodes, potentially granting them a higher perceived priority than equivalent pulled resources. Similarly, Safari does not (re-)prioritize pushed resources and keeps the default server-assigned weight (16, equal to font/XHR requests, see Table 2).

A bigger problem was identified for the Service Workers specification [14, 48], which allows developers to install a JS-based client-side proxy that can capture and amend egress resource requests [1, 7, 19], thus enabling various Web performance optimizations (such as split-browser setups). For both Chrome and Firefox, resource requests proxied through the Service Worker script become devoid of fine-grained HTTP/2 prioritization reasoning and are instead all grouped in the `High` priority bin or under the `followers` phantom node (with the default 22 weight value), respectively. Service Worker support for Safari is still in development.

Other techniques, such as `preload`/`prefetch` [47, 49] as well as `async` and `defer` JS [46] showed similar implementation deficiencies or unexpected behavior concerning HTTP/2 prioritization. Though it seems likely that these deficiencies will be mitigated in future User Agent releases, their current state might help explain why previous work has shied away from these Web technologies in favor of less standardized approaches.

5 EXPERIMENTAL SETUP

Our survey in Section 4 reveals that there is significant variation in User Agents' approaches to HTTP/2 prioritization. Even after source code review however, it remained unclear *why* the two most complex implementations, Chrome and Firefox, choose such orthogonal setups. Potentially their approach is simply a continuance of previous HTTP/1.x or SPDY heuristics, or possibly they have not yet experimentally determined optimal prioritization settings. We deem it more likely however that the differences reflect the characteristics of browser implementations as a whole (e.g., if Chrome is faster in a single-threaded setup it profits from a predominantly FCFS approach, while Firefox might opt for added parallelism because it is more optimized for multi-threading). Additionally, we hypothesized that (naive) RR will consistently hurt performance, as it allows contention among low- and high-priority assets, possibly delaying the latter category. We were therefore surprised that RR is implemented by several major browser vendors and promoted by the HTTP/2 specification.

5.1 Tested Schedulers

To validate our intuitions and assess the PLT performance impact of various prioritization approaches, we experimentally evaluated eight resource scheduling algorithms. Primarily, we compared the default HTTP/2 prioritization schemes of Chrome and Firefox (1) with the naive RR (2) and FCFS (3) schedulers, as they provide the two orthogonal extremes. We also included HTTPS/1.1 measurements (4), as here the same high-level prioritization heuristics are used, yet in a radically different way from HTTP/2 (HOL blocking on multiple connections, see Section 2.2.1). Finally, to assess how much Chrome's and Firefox's observed strategies are entangled with their overall implementation, we derived four flavors (5-8) of a scheduler that blends the HTTP/2 prioritization schemes of both User Agents. The four instantiations of this hybrid serial/parallel scheduling algorithm are described next.

5.1.1 Parallel+ for Chrome. Inspired by Firefox's tree-based setup, Parallel+ adds parallelism to Chrome (see Figure 3 (a)). High-priority resources are still transported serially, whereas medium- and low-priority assets are served in a parallelized fashion, as they can often take effect without having to be downloaded entirely (e.g., progressively renderable images). This setup is realized by introducing three top-level phantom nodes for grouping high-, medium- and low-priority requests. The left- and rightmost phantom nodes are statically granted the maximum and minimum HTTP/2 weight value, respectively. The weight value of the midmost phantom node is dynamically switched between these extremes depending on the presence of high-priority resources. In particular, the central clustering node will have a 256 weight value only when the leftmost branch of the HTTP/2 dependency tree is empty.

5.1.2 Serial+ for Firefox. Serial+ partly serializes resource delivery in Firefox (see Figure 3 (b)). More specifically, resources

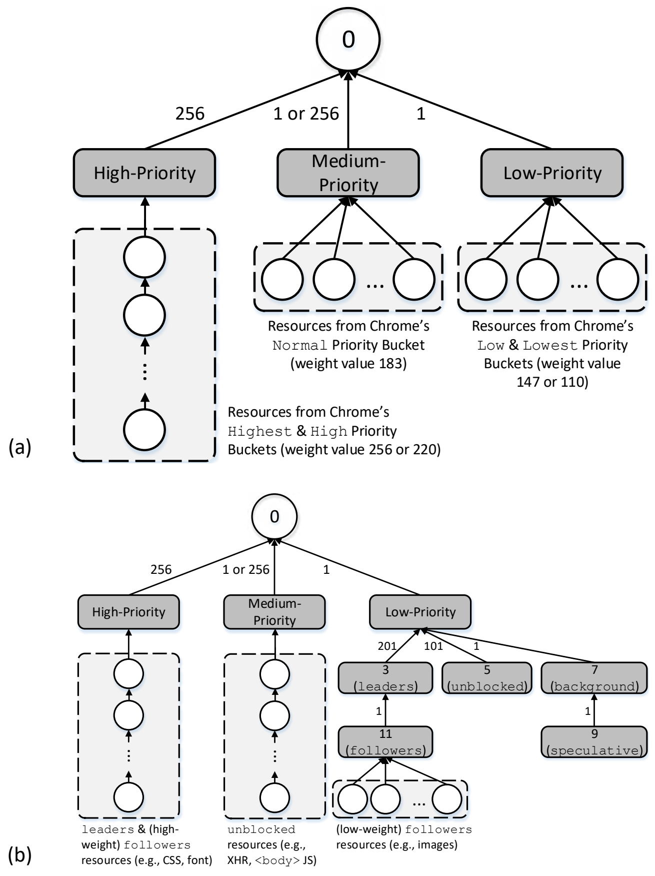

(a)
0
256
1 or 256
1
High-Priority
Medium-Priority
Low-Priority
Resources from Chrome's Normal Priority Bucket (weight value 183)
Resources from Chrome's Low & Lowest Priority Buckets (weight value 147 or 110)
Resources from Chrome's Highest & High Priority Buckets (weight value 256 or 220)
(b)
201
101
3 (leaders)
5 (unblocked)
7 (background)
11 (followers)
9 (speculative)
leaders & (high-weight) followers resources (e.g., CSS, font)
unblocked resources (e.g., XHR, <body> JS)
(low-weight) followers resources (e.g., images)

Figure 3: Drafting hybrid Parallel+/Serial+ HTTP/2 dependency trees for (a) Chrome and (b) Firefox.

from Firefox's leaders category (e.g., CSS, <head> JS) as well as followers resources with a weight larger than the default 22 value (e.g., HTML, font assets) are linearized under the leftmost phantom node via *exclusive* dependency relationships, while unblocked resources are made *exclusively* dependent on the midmost phantom node. The residual part of Firefox's original HTTP/2 dependency tree is then integrally preserved under the rightmost phantom node. The weights allotted to the phantom nodes are identical to those in the Parallel+ case, as is the logic to re-weight the midmost phantom node at run-time. For both Parallel+ and Serial+, all resource requests retain their original HTTP/2 weight value.

5.1.3 Simplified Parallel and Serial. Parallel+ and Serial+ actually represent the advanced flavors of the hybrid scheduling logic. In the "simplified" Parallel and Serial scenarios, the central phantom node is eliminated. Chrome's Normal priority resources are serially appended to their higher-priority counterparts under the auspices of the leftmost phantom node, while unblocked Firefox resources retain their position in the original dependency tree under the rightmost phantom node.

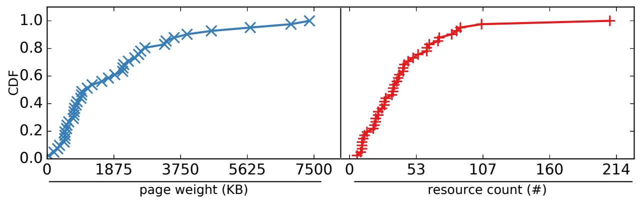


Figure 4: Cumulative Distribution Functions (CDFs) of page weight and asset count in our test corpus (n=40).

5.2 Apparatus and Evaluation Setup

To enact the discussed prioritization strategies, we extended version 2.1.0 of the H2O webserver [21]. Our server-side implementation ignores the client-issued prioritization directives and instead drafts a custom dependency tree according to the new scheduling logic, except of course when evaluating the default browser implementations. Note again that this is completely in accordance with the HTTP/2 specification [24]. We chose H2O because it is built and optimized for HTTP/2 from the ground up, as opposed to more popular server implementations which include HTTP/2 support as an additional plugin (e.g., NGINX, Apache). Previous work also shows H2O to be one of the most complete HTTP/2 capable servers [26].

We deployed our experimental setup using the Speeder framework [29], which integrates the WebPageTest tool [54] and exploits host virtualization to automate Web performance measurements across various (emulated) network conditions and configurable User Agents. As User Agents, we used versions 58 and 54 of respectively Chrome and Firefox, as these were the most stable versions compatible with WebPageTest at the time. These browsers were deployed on virtualized 64-bit Microsoft Windows 10 hosts, while H2O was run on 64-bit Linux Debian Jessie Virtual Machines (VMs). All VMs were accommodated on two Dell PowerEdge R420 machines, interconnected via 1 Gbps Ethernet, and were allocated at least 2 dedicated logical CPU cores and 2 GB RAM per VM.

The network emulations were performed using tc/netem and spanned both a Cable network link (30Mbps throughput, 40ms Round-Trip Time (RTT), no packet loss) and four cellular connection types with varying quality characteristics (termed Noloss, Good, Fair and Poor in order of degrading performance). The cellular connections were emulated by dynamically reenacting real-world mobile network performance traces previously used in Web performance research [16, 17]. Per example, Good has a median RTT of 39ms, median throughput of 7170Kbps and (very) short bursts of 1.7% to 73% packet loss (versus 86ms, 146Kbps and 2.7% to 85% for Poor).

5.3 Content Corpus

We evaluated the different prioritization methods on a total of 40 real-world webpages. Note that the HTTP/2 best practices [6, 20] indicate that sites should be distributed over as few servers as possible, thus making optimal use of HTTP/2's single TCP connection and maximizing the usefulness of the prioritization mechanism. We therefore cloned the pages with Wget [15] and served them locally from a single server. This approach is consistent with previous work [5, 27, 29, 35]. The tested pages are diverse in nature, ranging from news sites, over landing pages to product marketing content. The pages were selected from the Alexa Top 50 and Moz Top 500 rankings [2, 30] in such a way to obtain a good distribution of page weights and resource counts (see Figure 4).

6 EXPERIMENTAL RESULTS

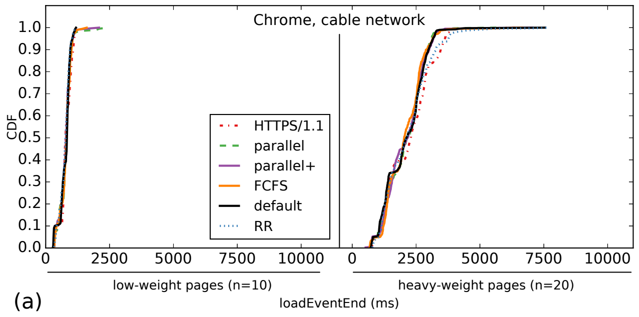
This section summarizes the principal outcomes of our full-factorial HTTP/2 prioritization benchmark (full set of results available at https://speeder.edm.uhasselt.be). We recorded multi-metric measurements from combining eight individual prioritization algorithms, two different User Agents, 40 pages and five network qualities into distinct experiments, each repeated 20 times. Due to space limitations, only the `loadEventEnd` and `SpeedIndex` results will be presented. The former metric is an indicator of the time it takes to load all the synchronous resources on a page [45], while the latter expresses the rate at which the page is populated with visual elements [55]. For both metrics, smaller values denote better PLT performance. Combined, these metrics provide a decent impression of both the quantitative and qualitative load performance of a webpage, though they are not necessarily strongly correlated with real user perceived performance [27, 43, 57]. While this can be considered a limitation of our study, conducting a full-factorial subjective evaluation of this magnitude is practically hard.

To facilitate comparison with prior work, we adopted the presentation and analysis method used by Bergan [5]. In particular, we applied the statistical Mann-Whitney U test with a very conservative p value of 0.005. Each metric was considered individually. When an experiment X yielded statistically significant differences compared to a *baseline* experiment Y for PLT metric Z (i.e., $p < 0.005$), we calculated the relative change of X compared to Y according to the following formula:

$$relative\ Z\ change = 1 - \frac{median(samples\ X\ expressed\ in\ Z)}{median(samples\ Y\ expressed\ in\ Z)}$$

These relative changes will be presented in tabular form. In these tables, cells highlighted in grey mark results referenced in the running text. For example, Table 3 offsets the performance of the naive approaches (RR and FCFS) to that of the default browser implementations as a baseline. For each combination of User Agent, PLT metric and prioritization case, the table lists per network configuration the quantity of testpages that were found to show no or small statistically significant differences with the baseline (reported in the R (Rest) column, ⩽5%), as well as the number of testpages that exhibited either a relative performance improvement or degradation of more than five and 25 percent (⩾5% and ⩾25% columns). For instance, from the first row in Table 3, we learn that in the `Cable` network setting no statistically significant `loadEventEnd` differences ⩾5% existed between the FCFS and default Chrome case for 24 out of the 40 tested webpages. On the other hand, two disjoint sets of eight webpages showed a statistically significant speedup or slowdown ⩾5%; three of the eight slower pages even incurred a penalty ⩾25% compared to the default.

As we have found page weight to be an important factor in the observed PLT trends, we will present not only results spanning the full corpus (in Tables 3, 4) but also yield split results for the 10 low-weight pages (total size ⩽600KB and/or resource count ⩽10) versus the 20 heavy-weight pages (total size ⩾1MB) in our corpus (in Figures 5, 6).



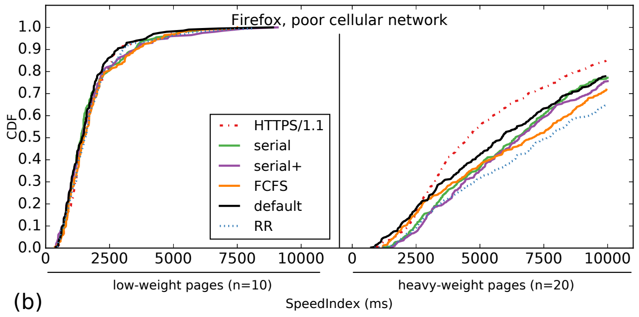


Figure 5: PLT performance CDFs for low-weight (left) versus heavy-weight (right) testpages.

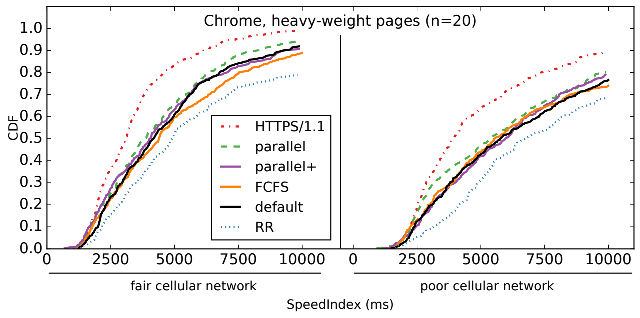


Figure 6: `SpeedIndex` CDFs for heavy-weight testpages in Chrome over `Fair` (left) and `Poor` (right) networks.

6.1 HTTP/2 Naive versus Default

We first look at the performance of the two naive methods, upon which the browsers' default implementations are expected to significantly improve. The results are summarized in Table 3, with the default browser implementations acting as the baseline case. We observe four important trends.

Firstly, as expected, the use of RR scheduling nearly unanimously led to slower page loads compared to the default algorithms. This finding manifests itself most prominently when considering the `SpeedIndex` metric, yet to a lesser extent also for `loadEventEnd`.

Table 3: Default browser HTTP/2 prioritization implementations (baseline) versus naive methods (n=40).

		Cable network					Good cellular network					Poor cellular network				
			speedup		slowdown			speedup		slowdown			speedup		slowdown	
setting	case	R	⩾5%	⩾25%	⩾5%	⩾25%	R	⩾5%	⩾25%	⩾5%	⩾25%	R	⩾5%	⩾25%	⩾5%	⩾25%
Chrome, `loadEventEnd`	FCFS	**24**	**8**	1	**8**	**3**	40	0	0	0	0	40	0	0	0	0
	RR	20	11	1	9	2	35	2	0	3	3	36	1	1	3	3
	vsFirefox	**3**	**37**	**15**	0	0	34	6	6	0	0	38	1	1	1	1
Chrome, `SpeedIndex`	FCFS	**28**	6	1	6	3	**35**	1	0	4	3	**37**	1	1	2	2
	RR	25	5	1	10	2	29	2	2	9	9	31	1	**1**	8	8
	vsFirefox	**4**	**35**	**17**	1	0	26	14	12	0	0	32	6	6	2	2
Firefox, `loadEventEnd`	FCFS	27	8	0	5	0	38	1	0	1	1	38	2	1	0	0
	RR	26	3	0	11	1	38	0	0	2	1	37	1	1	2	1
Firefox, `SpeedIndex`	FCFS	27	**6**	0	7	1	34	3	1	3	3	36	0	0	4	4
	RR	25	1	0	14	2	30	0	0	10	8	30	**1**	**1**	**9**	**9**

Consider for example Firefox on the `Poor` network quality; RR resulted in a significant `SpeedIndex` speedup for only a single webpage, whereas nine other webpages recorded significant slowdowns ⩾25%. Figures 5 (b) and 6 show that especially heavy weight pages' `SpeedIndex` deteriorates on RR. The largest slowdowns (up to 214%) were due to very deep dependency chains (e.g., a JS file uses XHR to fetch a JSON file which includes image URLs).

Secondly, the FCFS and default schedulers were roughly balanced in the `Cable` network setting (i.e., the number of page load speedups and slowdowns is similar), whereas this balance tended to slightly tip in favor of the default approaches as network conditions deteriorated. Additionally, the quantity of differences ⩾25% were limited, especially when compared to RR. We initially interpreted the surprisingly good performance of the naive FCFS approach to be an indication of User Agents' intelligence for re-ordering resource requests (e.g., using a heuristics-based speculative HTML parser, see Section 2.2.1). However, when trying to confirm this thesis via a *lexical* scheduler (i.e., one that delivers resources in the exact order they are declared in the HTML), the results showed a comparable level of performance (results omitted due to space constraints). We therefore tentatively infer that the high FCFS and lexical performance might be explained by the optimized HTML code of the tested popular webpages, which likely underwent manual subresource declaration order optimization.

Thirdly, the quantity of statistically relevant PLT performance differences decreased nearly monotonically as network link quality degraded. As an example, for Chrome on `Cable`, statistically significant `SpeedIndex` differences ⩾5% were recorded between the FCFS and default implementation for 30% (12 out of 40) of the webpage corpus; this amount was more than halved on `Good` cellular and further dropped to 7.5% when network quality was most impaired. This shows that the browsers' default behaviors often only scarcely outsmart naive algorithms on impaired networks, which helps explain why prior work is able to achieve impressive speedups on mobile networks [31, 36, 53].

Fourthly, for low-weight pages, the PLT performance differences became marginal across network conditions (Figure 5 left side). It seems that Web transport optimization for low-weight webpages is subject to the law of diminishing returns.

Interestingly, a number of unexpected anomalies could be explained by specific page setups. For example, the single site with ⩾25% faster `SpeedIndex` on RR for Chrome on the `Poor` network consisted of many smaller images which had a large impact on the final page render, while it also encompassed large JS files with little visual effect. On RR, Chrome could start rendering much sooner than under the default prioritization behavior. A second page included a number of large images, of which only one was visible "above the fold". Using FCFS on Firefox caused this "hero image" to be downloaded before the other images, instead of being interleaved with them, leading to 24% faster `SpeedIndex` on `Cable`. These results showcase that browsers' tendency to coarsely group resources by type (Section 4) can profit from additional manual control [56].

Finally, we also compared the default HTTP/2 prioritization implementations of Firefox and Chrome directly in Table 3 (vsFirefox). The results show that Firefox is as a whole profoundly faster than Chrome, especially on faster networks. It is however impossible to assess how much of this superior performance is due to Firefox's more complex HTTP/2 prioritization setup, as the two browsers also differ significantly in almost all aspects of their underlying engine.

6.2 Parallel(+), Serial(+) and HTTPS/1.1

Recall that Parallel(+) and Serial(+) should help us assess whether the browsers' default HTTP/2 prioritization implementations were chosen specifically to work in symphony with their overall underlying engines. Unexpectedly, the results in Table 4 show no obvious trends. For some settings, parallelization improved performance in Chrome (e.g., by ⩾25% on `Good`, `SpeedIndex`), as does serializing assets in Firefox (e.g., on `Cable` for `loadEventEnd`, though only by ⩾5%). On the other hand, situations exist where these schedulers incurred a PLT performance penalty compared to the default behaviors and there is no consistent winner between the simple and advanced (+) versions.

Especially for Parallel(+) it came as a surprise that the added interleaving of image assets did not lead to an improved `SpeedIndex`, as Chrome can typically progressively render multiple images at the same time (e.g., when they are downloaded on multiple HTTP/1.x connections). For the 20 heaviest testpages (17 of which incorporate a decent quantity of images) in Figure 6, simplified parallelization in fact did seem to yield (mostly statistically insignificant) `SpeedIndex` gains for the two most impaired network configurations, without hereby substantially impacting `loadEventEnd` performance.

Table 4: Default browser implementations (baseline) versus Parallel(+)/Serial(+) variations and HTTPS/1.1 (n=40).

		Cable network					Good cellular network					Poor cellular network				
			speedup		slowdown			speedup		slowdown			speedup		slowdown	
setting	case	R	⩾5%	⩾25%	⩾5%	⩾25%	R	⩾5%	⩾25%	⩾5%	⩾25%	R	⩾5%	⩾25%	⩾5%	⩾25%
Chrome, loadEventEnd	Parallel	22	7	0	11	1	39	0	0	1	1	36	2	2	2	2
	Parallel+	19	9	1	12	2	38	0	0	2	2	37	2	2	1	1
	HTTPS/1.1	17	7	1	**16**	**5**	37	1	1	2	1	35	**5**	**4**	0	0
Chrome, SpeedIndex	Parallel	29	3	0	8	2	34	**5**	**4**	1	1	35	3	3	2	2
	Parallel+	31	4	1	5	0	36	2	2	2	1	34	3	3	3	3
	HTTPS/1.1	24	3	2	**13**	**4**	36	3	3	1	1	32	**8**	**7**	0	0
Firefox, loadEventEnd	Serial	26	**8**	0	6	1	36	0	0	4	3	38	0	0	2	2
	Serial+	22	**12**	0	6	2	38	0	0	2	1	37	1	0	2	2
	HTTPS/1.1	18	2	0	**20**	**4**	34	3	1	3	3	32	**8**	**8**	0	0
Firefox, SpeedIndex	Serial	26	10	1	4	1	34	1	1	5	5	37	0	0	3	3
	Serial+	25	6	0	9	1	35	1	1	4	4	35	2	2	3	3
	HTTPS/1.1	16	4	0	**20**	**5**	31	4	3	5	4	35	**3**	**3**	2	2

Finally, our results for HTTPS/1.1 demonstrate that HTTP/2 significantly outperformed HTTPS/1.1 in the highest-quality network setting (i.e., Cable), while HTTPS/1.1 decisively took the upper hand for Poor. This is most likely attributable to HTTPS/1.1's use of multiple TCP connections, which adds overhead on fast networks but offers robustness in the case of packet loss (as corroborated by previous work [10, 11, 18, 29, 52], which indicates that HTTP/2 could also benefit from using additional TCP connections on slower networks).

7 DISCUSSION AND CONCLUSIONS

Our survey of contemporary User Agents in Section 4 is the first to firmly showcase large discrepancies in how they approach webpage resource prioritization and in how this is consequently mapped to the HTTP/2 protocol. However, we were unable to find conclusive proof that these distinct schemes are fully optimized for the peculiarities of the underlying browser engines. While some results (e.g., for Parallel(+)/ Serial(+), and the slowdowns on RR) could be accredited to this effect, there are also counter-arguments, such as the fact that the browsers' built-in schedulers do not always improve on naive HTTP/2 prioritization algorithms, especially for low-weight pages.

In our opinion, this can have two main reasons. First, various User Agents might not yet be (fully) optimized with respect to HTTP/2's prioritization system (and/or prioritization in general). We judge this is most likely the case for IE, Edge and Safari (and to a lesser extent for Chrome and Firefox for techniques like Push and Service Workers, see Section 4.4) and expect them to evolve their approach over time. The other reason (which was also hypothesized in related work [31, 52]) is that it is non-trivial to improve PLT purely via generic network-level prioritization due to the complex interplay that exists with other parts of the browser engine. This seems to be the primary case for Firefox and Chrome and would also explain why advanced re-prioritization techniques [31, 36, 53] often need to optimize priorities for both computation and networking aspects on a per-page basis to obtain substantial PLT gains (and why our generic Parallel(+) and Serial(+) failed to perform as expected). However, given the large internal differences between browser engines, we put forward that these re-prioritization systems should also take into account the User Agent to achieve optimal results, which to date none seem to do.

In all, we conclude that (HTTP/2) prioritization can indeed influence Web performance (and hence user experience), yet that careful alignment with other components of the webpage load process (e.g., HTML optimization, browser internals) is needed, likely on a per-page basis, to warrant PLT speedups. Additionally, in our results RR performs worst and thus should perhaps not be recommended by the HTTP/2 specification.

Fortunately, the HTTP/2 prioritization standard has proven to be versatile enough to afford complex resource scheduling logic as well as run-time (re-)prioritization. As such, the advanced per-page prioritization accelerators of the future can make use of this standard setup instead of having to circumvent the browser with custom methods. In the short term, this can be achieved by drafting custom dependency trees at the server (as shown by our evaluation), while future Web standards (such as Service Workers [14] and Priority Hints [56]) will allow developers to influence priorities emitted by the client directly. Interesting future work lies in analyzing why the FCFS scheduler performed so well and in looking at cross-connection prioritization, either when sharding HTTP/2 over multiple domains or with the QUIC protocol [23].

8 ACKNOWLEDGMENTS

Part of this work was funded by the imec ICON Pro-Flow project. Industrial partners involved in the Pro-Flow project include ANDROME, Nokia and Barco. Robin Marx is a SB PhD fellow at FWO, Research Foundation - Flanders, project number 1S02717N.

REFERENCES

[1] Addy Osmani. 2017. The PRPL Pattern. Online, https://w3c.github.io/webvr/spec/latest/. (May 2017).
[2] Alexa. 2017. Top 500 Global Sites. Online, http://www.alexa.com/topsites. (2017).
[3] Andy Davies. 2013. How the Browser Pre-loader Makes Pages Load Faster. Online, https://andydavies.me/blog/2013/10/22/how-the-browser-pre-loader-makes-pages-load-faster/. (2013).
[4] Awio Web Services LLC. 2017. Web Browser Usage Trends. Online, https://www.w3counter.com/trends. (2017).
[5] Tom Bergan. 2016. Benchmarking HTTP/2 Priorities. Online, https://docs.google.com/document/d/1oLhNg1skaWD4DtaoCxdSRN5erEXrH-KnLrMwEpOtFY/. (October 2016).

[6] Tom Bergan, Simon Pelchat, and Michael Buettner. 2016. Rules of Thumb for HTTP/2 Push. Online, https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/edit. (August 2016).

[7] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. 2017. Progressive Web Apps: The Possible Web-native Unifier for Mobile Development. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST*. SciTePress, 344–351.

[8] Browserscope. 2017. Max connections per browser. Online, http://www.browserscope.org/?category=network. (October 2017).

[9] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. 439–453.

[10] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 609–614.

[11] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. 2015. Is HTTP/2 really faster than HTTP/1.1?. In *Proceedings of the IEEE Conference on Computer Communications Workshops*. 293–299.

[12] Dev.Opera. 2012. Opera Mini and JavaScript. Online, http://dev.opera.com/articles/view/opera-mini-and-javascript/. (September 2012).

[13] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and K. K. Ramakrishnan. 2013. Towards a SPDY'ier Mobile Web?. In *Proceedings of the 9th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT'13)*. 303–314.

[14] Matt Gaunt. 2017. Service Workers: an Introduction. Online, https://developers.google.com/web/fundamentals/getting-started/primers/service-workers. (May 2017).

[15] GNU Project. 2017. Wget. Online, https://www.gnu.org/software/wget/. (2017).

[16] Utkarsh Goel. 2016. A script based on TC netem to emulate the latency, loss, and bandwidth of a real-world cellular network. Online, https://github.com/akamai/cell-emulation-util. (2016).

[17] Utkarsh Goel, Moritz Steiner, Mike P. Wittie, Martin Flack, and Stephen Ludin. 2016. Poster: HTTP/2 Performance in Cellular Networks. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom'16)*. 433–434.

[18] Utkarsh Goel, Moritz Steiner, Mike P Wittie, Stephen Ludin, and Martin Flack. 2017. Domain-Sharding for Faster HTTP/2 in Lossy Cellular Networks. *arXiv preprint arXiv:1707.05836* (2017).

[19] Utkarsh Goel, Mike P Wittie, and Moritz Steiner. 2015. Faster Web through Client-assisted CDN Server Selection. In *Computer Communication and Networks (ICCCN), 2015 24th International Conference on*. IEEE, 1–10.

[20] Ilya Grigorik. 2013. *High Performance Browser Networking: What every web developer should know about networking and web performance.* " O'Reilly Media, Inc.".

[21] H2O. 2017. The optimized HTTP/1.x, HTTP/2 server. Online, https://h2o.examp1e.net/. (2017).

[22] Bo Han, Shuai Hao, and Feng Qian. 2015. MetaPush: Cellular-Friendly Server Push For HTTP/2. In *Proceedings of the Workshop on All Things Cellular: Operations, Applications and Challenges (AllThingsCellular'15)*. 57–62.

[23] IETF Network Working Group. 2016. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. Online, https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02. (January 2016).

[24] IETF RFC7540. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). Online, https://tools.ietf.org/html/rfc7540. (May 2015).

[25] Ilya Grigorik, Pat Meenan, Rick Viscomi. 2017. HTTP Archive. Online, http://httparchive.org/. (October 2017).

[26] Muhui Jiang, Xiapu Luo, Tungngai Miu, Shengtuo Hu, and Weixiong Rao. 2017. Are HTTP/2 Servers Ready Yet?. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 1661–1671.

[27] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. 2017. Improving User Perceived Page Load Times Using Gaze.. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 545–559.

[28] Ron Kohavi, Alex Deng, Roger Longbotham, and Ya Xu. 2014. Seven rules of thumb for web site experimenters. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1857–1866.

[29] Robin Marx, Peter Quax, Axel Faes, and Wim Lamotte. 2017. Concatenation, Embedding and Sharding: Do HTTP/1 Performance Best Practices Make Sense in HTTP/2?. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST'17)*. 160–173.

[30] Moz. 2017. Top Sites: The 500 Most Important Websites on the Internet. Online, https://moz.com/top500. (2017).

[31] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI'16)*. 123–136.

[32] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP.. In *USENIX Annual Technical Conference*. 417–429.

[33] Shubhie Panicker. 2016. Hero Element Timing API. Online, https://docs.google.com/document/d/1yRYfYR1DnHtgwC4HRR04ipVVhT1h5gkI6yPmKCgJkyQ. (September 2016).

[34] Radware. 2016. Web Performance State of the Union. Online, https://www.radware.com/social/industry-sotu2016/. (June 2016).

[35] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. 2017. Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 459–468.

[36] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. 2017. VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 390–403.

[37] Ali Sehati and Majid Ghaderi. 2015. WebPro: A proxy-based approach for low latency web browsing on mobile devices. In *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*. IEEE, 319–328.

[38] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. 2014. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of the 10th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT'14)*. 325–336.

[39] StatCounter Global Stats. 2017. Browser, OS, Search Engine including Mobile Usage Share. Online, http://gs.statcounter.com/browser-market-share/mobile/worldwide/#monthly-201709-201709-bar. (2017).

[40] Tammy Everts, Tim Kadlec. 2017. WPO Stats. Online, https://wpostats.com/. (October 2017).

[41] The Chromium Projects. 2014. SPDY Protocol. Online, https://www.chromium.org/spdy/spdy-protocol. (2014).

[42] Anne van Kesteren. 2017. XMLHttpRequest Living Standard. Online, https://xhr.spec.whatwg.org/. (May 2017).

[43] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. 2016. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of the 12th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT'16)*. 399–412.

[44] Matteo Varvello, Kyle Schomp, David Naylor, Jeremy Blackburn, Alessandro Finamore, and Konstantina Papagiannaki. 2016. Is the Web HTTP/2 Yet?. In *International Conference on Passive and Active Network Measurement*. Springer, 218–232.

[45] W3C Recommendation. 2012. Navigation Timing. Online, https://www.w3.org/TR/navigation-timing. (December 2012).

[46] W3C Recommendation. 2014. HTML5 - A vocabulary and associated APIs for HTML and XHTML. Online, https://www.w3.org/TR/html5/. (October 2014).

[47] W3C Working Draft. 2016. Preload. Online, https://www.w3.org/TR/preload/. (November 2016).

[48] W3C Working Draft. 2016. Service Workers 1. Online, https://www.w3.org/TR/service-workers-1/. (October 2016).

[49] W3C Working Draft. 2017. Resource Hints. Online, https://www.w3.org/TR/resource-hints/. (May 2017).

[50] Carl A. Waldspurger and William E. Weihl. 1994. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI'94)*. Article 1.

[51] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. 473–486.

[52] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How Speedy is SPDY?. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 387–399.

[53] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI'16)*. 109–122.

[54] WebPagetest. 2017. Website Performance and Optimization Test. Online, https://www.webpagetest.org/. (2017).

[55] WebPageTest Documentation. 2012. Speed Index. Online, https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index. (2012).

[56] Addy Osmani Yoav Weiss. 2017. Priority Hints. Online, https://github.com/WICG/priority-hints. (August 2017).

[57] Torsten Zimmermann, Benedikt Wolters, and Oliver Hohlfeld. 2017. A QoE Perspective on HTTP/2 Server Push. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*. ACM, 1–6.