Is the Web ready for HTTP/2 Server Push?

Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University
{zimmermann,wolters,hohlfeld,wehrle}@comsys.rwth-aachen.de

ABSTRACT

HTTP/2 supersedes HTTP/1.1 to tackle the performance challenges of the modern Web. A highly anticipated feature is Server Push, enabling servers to send data without explicit client requests, thus potentially saving time. Although guidelines on how to use Server Push emerged, measurements have shown that it can easily be used in a suboptimal way and hurt instead of improving performance. We thus tackle the question if the current Web can make better use of Server Push. First, we enable real-world websites to be replayed in a testbed to study the effects of different Server Push strategies. Using this, we next revisit proposed guidelines to grasp their performance impact. Finally, based on our results, we propose a novel strategy using an alternative server scheduler that enables to interleave resources. This improves the visual progress for some websites, with minor modifications to the deployment. Still, our results highlight the limits of Server Push: a deep understanding of web engineering is required to make optimal use of it, and not every site will benefit.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Network measurement**;

KEYWORDS

HTTP/2, Server Push, Interleaving Push

ACM Reference Format:
Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, Klaus Wehrle. 2018. Is the Web ready for HTTP/2 Server Push?. In *CoNEXT '18: International Conference on emerging Networking EXperiments and Technologies, December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3281411.3281434

1 INTRODUCTION

The Hypertext Transfer Protocol (HTTP) is the de-facto protocol for realizing desktop and mobile websites as well as applications. Traffic shares of > 50 %, e.g., in a residential access link [24], an IXP [6], or backbone [12, 33], express this dominance. Despite this, HTTP-based applications are built on top of a protocol designed nearly two decades ago, now suffering from various inefficiencies in the modern Web, e.g., Head of Line (HoL) blocking. To address the drawbacks, HTTP/2 (H2) was standardized [8] as H1's successor. Among others, a highly anticipated feature [17] is *Server Push*, changing the *pull-only* into a *push-enabled* Web. It enables servers to send additional resources *without* explicit requests, e.g., send a CSS upon a request for `index.html`, thus saving round trips.

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

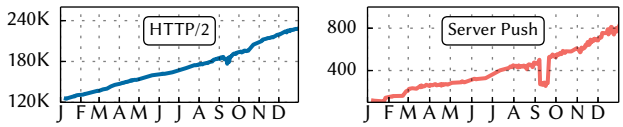


Figure 1: Adoption of HTTP/2 and Server Push over the course of one year[1](2017) on the Alexa 1 M list [39]. Although the use of both is increasing, the adoption of Server Push is relatively low.

This potential for speeding up the Web manifests in a growing interest in Server Push, both among CDNs [10, 38] and in research [14, 22, 32, 39]. These efforts face the challenge that the standard only defines the Server Push protocol, not how to use it, i.e., *what* to push *when*, determining its performance. In this regard, previous work provided several strategies or approaches how to use Server Push. They involve signaling clients what to fetch next [32], dependency analysis of content [14], gaze tracking to identify regions of interest to be pushed [22], or guidelines for its basic usage [10]. However, its usage is still low relative to the H2 adoption (cf. Fig. 1) [35, 39]—potentially given its complex usage. In previous studies, we showed that Server Push can be easily used suboptimally in real-world deployments and *hurt* instead of *improving* performance [39, 40]. These findings, as well as discussions among web and protocol engineers [7, 23, 26, 28], highlight that the quest for *optimal* Server Push usage is far from being settled.

We thus tackle the question *what* influences the Server Push performance, *how* push strategy performance can be reproducibly tested on any website, and *how* a new strategy can speed up the current Web without major modifications. Specifically, we *i)* propose a new evaluation method to automatically and reproducibly evaluate Server Push strategies on any website by replay. This complements measurements of real-world deployments and provides a method to systematically understand the isolated baseline performance of Server Push strategies for a broad class of websites. We *ii)* use this method to study the performance impact of strategies on real-world and synthetic websites. In this regard, we revisit *existing* strategies proposed in related work and find that pushing all objects, as a straightforward approach, can hurt the performance. By varying the amount, order, and type of objects, we observe positive *and* negative effects, highlighting the challenge of optimal usage. Given these results and based on an analysis of the respective rendering process in the browser, we *iii)* investigate a new approach to push the *right resource* at the *right time*, by interleaving the base document and pushed objects. This *can* lead to a faster visual progress for some popular websites. Still, our results also highlight the limits of Server Push, as it requires a deep understanding of the load and render process and not every website can benefit.

[1]Live results available at https://netray.io

2 BACKGROUND

We now briefly present an overview of H2, as well as two prominent metrics to assess the performance of websites.

2.1 H2 Overview

H2 was standardized in 2015 [8] to replace H1. A key difference is a binary instead of an ASCII representation, allowing easier framing and parsing, which increases the processing efficiency [17]. Further, H2 allows to *multiplex* requests/responses over *one* connection, leading to parallel *streams*, identified by IDs. Clients can *prioritize* streams, e.g., to prefer certain objects. Multiplexing also reduces application layer HoL blocking, an issue in H1. Like H1, H2 is *stateless*, thus header information for a connection is repetitive. As mitigation, H2 adds *header compression* [30]. Though not mandatory, most implementations use H2 via TLS. Besides privacy and security benefits, a TLS-tunnel over deployed intermediaries eases the deployment of new protocols [17].

H2 Server Push. Finally, H2 adds *Server Push*. To grasp its potential benefits, recall the classic H1 request/response model: a browser requests the base document, parses it, and then requests all discovered objects individually. Contrary, an H2 server can *push* objects without *explicit* request, e.g., a CSS upon a request to `index.html`, thus saving round trips. Hence, H2 allows transferring resources *before* the browser finishes parsing. To push, a server announces information about the object and stream it will use. Afterward, the data is sent. A server is only allowed to push content from origins under its authority. Further, a client can cancel an announced push, which is useful if the object is already cached. Also, a client can deactivate the feature by setting `SETTINGS_ENABLE_PUSH` in a specific settings header to `0` at connection startup. Yet, as seen in own measurements, by the time a client cancels the push, the object can be already in flight. Also, the standard does not include mechanisms to signal the cache status, but drafts and academic approaches exist [20, 29].

2.2 Website Performance Metrics

PLT. In recent years, the key metric to measure performance was, and still is, Page Load Time (PLT) [14, 22, 31, 35–37], which represents the time between events in the browser's W3C Navigation Timing API. Mostly, it refers to the time of the `onload` event, i.e., a resource and its dependent resources finished loading, but other events are used in related work as well. Here, we define PLT as the time between the `connectEnd` event, i.e., the connection is established (DNS, TCP and TLS) and start of the `onload` event. Still, PLT can be an over- or underapproximation [22], e.g., events may refer to items not in view, and some resources can be loaded by scripts *afterward*. Hence, PLT can fail to capture human perception [11, 40].

SpeedIndex. To overcome the limits of PLT, Google proposed SpeedIndex [5] to capture the visual progress of *above-the-fold* content, i.e., content in the viewport without scrolling. It expresses how *complete* a website looks at various points of its loading process. To calculate SpeedIndex, the loading process is recorded as a video and each frame is compared to the final frame, thus measuring completeness. While this *visual* metric is an improvement over PLT, capturing a video may only be feasible for studies in the lab but not in the wild [11, 25].

3 RELATED WORK

Our work relates to approaches that focus on H2 and Server Push performance, as well as frameworks and guidelines.

H2. Wang et al. [37] provided the first analysis of SPDY, H2's predecessor. Comparing SPDY to H1 on the transport level, they observe benefits, especially for large objects and low loss, and for few small objects in good conditions with large TCP initial windows. Also, Server Push *can* improve PLT for large RTTs, but the analysis of several policies also reveals impairments. De Saxcé et al. [15] evaluate H2 and focus on latency and loss, showing that H2 is less prone to higher latencies than H1. They regard Server Push as valuable, but more performance research is required. Varvello et al. [35] present an adoption study (Alexa 1 M) and compare the real-world performance of H2 websites to their H1 counterparts. They observe benefits for 80 % of websites, but also degradations, without an explicit focus on Server Push.

Server Push. In previous work [39], we complement the view on H2's adoption provided in [35] and target broader sets, i.e., IPv4 scans and all .com/.net/.org domains, and Server Push explicitly. While the adoption of H2 *and* Server Push is rising, the latter is orders of magnitude lower (cf. Fig. 1). Using different protocol settings, we observe that Server Push can improve as well as hurt the performance, but the results cannot be mapped to simple reasons (e.g., amount of bytes pushed), and that further analysis is required. Instead of resources, Han et al. [20] propose to push *hints*, enabling the client to request critical resources earlier, which *can* improve the performance. Rosen et al. [31] analyze the benefits and challenges of Server Push and show that network characteristics play a major role in the effectiveness, similar to [15, 37]. As one guideline, they propose to push as much as possible, which not *always* leads to improvements. Butkiewicz et al. [14] present Klotski, which prioritizes high-utility resources, e.g., above-the-fold or by user preferences, obtained via surveys, live ratings, and offline measurements. To deliver those, Server Push is used. Kelton et al. [22] prioritize objects of visual interest, identified in user studies via gaze tracking. As objects can depend on each other, they employ dependency analysis [36] to prioritize objects and send high priority objects via Server Push. Still, as there are impairments for some websites, they revert to the default setting in such cases. Vroom [32] uses a client scheduler that parses *preload* headers [18], containing dependency hints for resources (on other servers) to be fetched. As in [14], high priority resources are pushed. This combination can improve performance compared to base H2. In parallel to our work, results for the real-world performance of Server Push presented at the IETF [23, 26] indicated that more measurements are needed to grasp its benefits and even provoked the discussion, if Server Push should be focused on in the future and be used at all.

Rules of Thumb for Push. Bergan et al. [10] provide a set of rules, evaluated in different simulations, network settings, and websites. We focus on rules relevant for this paper and refer to [10] for more information. A server should *i)* push just enough to fill idle network time. As soon as the browser parses the HTML and requests objects, using push is not faster. This is contrary to [31], suggesting to push as much as possible. Further, objects should *ii)* be pushed in evaluation-dependent order, as suboptimal orders can e.g., delay the discovery of hidden resources loaded by scripts.

Takeaway: Despite existing frameworks and guidelines, H2 Server Push is still not widely in use, and if, improvements are not guaranteed, as shown by recent measurements of real-world deployments. Our goal is to provide an approach to understand the isolated performance of Server Push systematically and to shed light on its applicability for the current Web, i.e., *without* major modifications.

4 REPLAYING PUSH WEBSITES

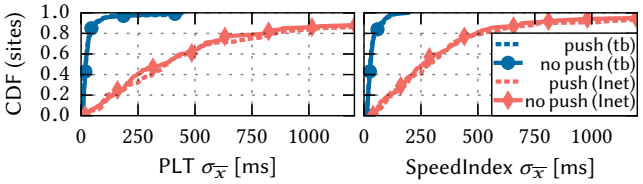
Although assessing the performance of Server Push in real-world deployments [23, 26, 35, 39] is crucial, it imposes practical challenges. Among other factors, websites *i)* can change due to dynamic third-party content, e.g., ads [16], or *ii)* are subject to varying network characteristics. This can cause misinterpretations of the results. Hence, our goal is to understand the *isolated* performance of Server Push under deterministic conditions, to reduce variability, and to enable reproducibility. Therefore, we use a testbed to replay real-world websites and to grasp the possible potential of Server Push. We exemplify this by using *different* strategies and revisit existing guidelines, e.g., [10, 31], to assess their overall impact. Next, we present our testbed and evaluation of different strategies.

4.1 Controlled Push Strategy Evaluation

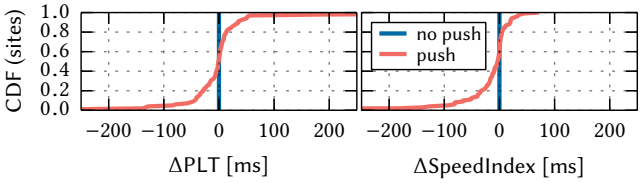
We base our testbed, which we make publicly available together with more results [9], on Mahimahi [27]. This enables to record H1 traffic to a database as request/response pairs, e.g., captured in a browsing session. Later, this database matches requests to responses to replay websites. Network namespaces are used to recreate the deployment, i.e., for each IP a local server is spawned. Thus, the same connection pattern as in the Internet is used, opposed to less realistic setups, such as local mirroring (e.g., HTTrack) or proxies.

However, at the time of writing, the current version of Mahimahi does not support H2. To enable this support in Mahimahi, we first use the H2-capable `mitmproxy` [3] to capture request/responses and convert it to Mahimahi's record format. We add the `h2o` webserver [2] and create an `h2o`-FastCGI module that matches and serves responses from the record DB. Finally, we enable to specify *push strategies*, to define responses to be pushed. H2 supports connection-coalescing, i.e., a server with a single IP can be authoritative for multiple domains and serve content via *one* connection. Currently, a browser handles traffic for different origins over the *same* connection if a server presents a TLS certificate that includes the origins as *Subject Alternative Names*. The browser also checks if IPs of different origins match using DNS. We thus modify Mahimahi to generate certificates for *each* local server, which include domains with the same IP. Also, we assume *every* server to be H2-enabled, as in [32]. Using `tc`, we simulate DSL settings with 50 ms RTT and 16 Mbit/s down- and 1 Mbit/s uplink between the client and the servers. Please note that we do not assume any additional delay *on* the servers, e.g., for additional fetches or disk access. For all evaluation settings, we follow settings used in previous work [40] and utilize `browsertime` [1] to automate Chromium 64 and replay each website in each setting for 31 times. If not stated otherwise, we show and discuss the median result of these repeated runs.

Evaluation. We evaluate our testbed by replaying 100 random websites using Server Push from Alexa 1 M, with and without Server Push. Fig. 2(a) shows the standard error for PLT and SpeedIndex



(a) Comparing the std. error $\sigma_{\overline{x}}$ of the performance of 100 websites in our testbed (tb) vs. the Internet (Inet) over 31 runs.



(b) In our testbed, we observe similar effects as in the Internet, i.e., improvements and detriments with Server Push when compared to the *no push* case (Δ < 0 is better).

Figure 2: Evaluation of our realized testbed based on Mahimahi.

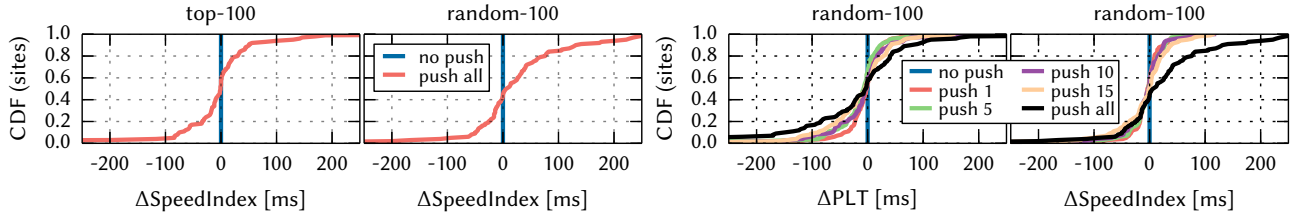
(cf. Sec. 2.2), compared to their deployment in the Internet. For 95 % (85 %) of websites, the error $\sigma_{\overline{x}}$ is < 100 ms (50 ms) for PLT, with similar results for SpeedIndex. In the Internet, this holds only for 14 % (5 %), which we attribute to varying network conditions. These results indicate that we remove a lot of variability for the performance results of most websites, which helps to assess the performance impact in a reproducible manner. To evaluate if we still, after removing this variability, observe performance improvements and impairments as the Internet [39], we compare the performance of Server Push against a *no push* configuration, i.e., the client signals the server to disable Server Push (cf. Sec. 2.1). Pushing the *same* objects as in the Internet, we observe no benefits for 49 % (35 %) of websites in PLT (SpeedIndex) (cf. Fig. 2(b)). We thus argue that our testbed enables to reproduce results for a lot of websites and we still observe positive *and* negative effects (cf. Sec. 3).

4.2 Real-World: Altering What to Push

Next, we replay real-world websites subject to various push strategies in our testbed, which enables us to study the impact on performance in a *controlled* environment. To this end, we create two disjunct random sets of 100 websites (HTTPS) each, one from the top 500 (top-100) and from the top 1 M (random-100) according to Alexa. We use the Alexa list as a basis, as we expect a lot of H2-enabled websites among this popular list [34, 39]. If there is *no* H2 version, we capture the respective H1 version. Please note that in case of an H2-enabled website, we do not capture if the website uses Server Push, as our goal is to apply *different* strategies.

Pushable Objects. For the top-100 (random-100) set, 52 % (24 %) have < 20 % of pushable objects, i.e., the other objects reside on servers beyond the authority of the pushing server. Hence, many websites *cannot* push all objects.

Computing the Push Order. Next, we access the websites in our testbed via H2 31 times, not pushing *any* objects. We trace requests and priorities (cf. Sec. 2) used to obtain the landing page and construct a dependency tree, based on these priorities. By traversing this tree, we compute a request order. The goal is to obtain the desired order used by the browser to request objects, to



(a) SpeedIndex when pushing all objects (in request order) normed to the *no push* case. PLT shows a similar behavior.

(b) Push limited amount, normed to the *no push* case. Only for the random set, due to limited objects for sites in the top set.

Figure 3: Delta (median) when pushing all (Fig. 3(a)) and a limited amount of objects (Fig. 3(b)), using a computed order.

identify an order to push. Here, we follow the rules given in [10] (cf. Sec. 3), as suboptimal orders can have negative impacts, e.g., delay critical resources. Since the order is not stable across all runs, e.g., due to client-side processing, we use a majority vote. As this order is based on the *initial* connection to the origin server, all objects are pushable (cf. Sec. 2.1).

4.2.1 Varying Amount and Type of Objects. First, we push all *pushable* objects following the computed order, as *push all* is considered a valuable strategy in [31]. Fig. 3(a) shows the results for SpeedIndex. In the top-100 (random-100) set only 58 % (45 %) of sites benefit (PLT similar but not shown). Pushing *all* objects can be harmful, as it can delay processing, and even in the positive case, waste bandwidth or cause contention on the server [20, 32]. Thus, we vary the amount of objects $n \in \{1, 5, 10, 15\}$ to push. Here, we only consider the random-100 set, as not all top-100 websites have enough pushable resources. We again use fixed orders, limited to the first n objects. For a small n, this is in line with guidelines [10] (cf. Sec. 3) to push just enough to fill the network idle time. We observe (cf. Fig. 3(b)) that pushing less can lead to less detrimental effects, e.g., delay of processing and requests. Still, a lot of websites exhibit no *significant* improvements.

Last, we analyze the impact of pushing specific object types, again for the random-100 set. While pushing CSS or JavaScripts (JS) can lead to positive and negative effects (no figure shown), images lead to a *worse* SpeedIndex for 74 % of websites. This is no surprise, as they do not contribute to the creation of the Document- (DOM) or CSS Object Model (CSSOM)—both essential parts of the layout and render process. Using the *best* type strategy per website, i.e., if pushing CSS is better than JS, CSS is the best type strategy, only 24 % (SpeedIndex) and 20 % (PLT) of websites improve. Type combinations, i.e., CSS+JS and CSS+images, lead to similar results.

Also, we analyzed the effect if we vary the computed push order (results not shown). We observe that the impact on performance is highly dependent on the overall amount of objects, the structure of the HTML, i.e., when objects are referenced, and object type. Exemplary, a suboptimal order could prefer uncritical resources, with respect to above-the-fold, and thus delay critical resources.

Conclusion: We observe that a large fraction of resources is not pushable as they reside on other servers. Moreover, many websites depend on third-party content—impacting the loading process [13, 16]. Also, not many sites benefit from a *push all* strategy. Pushing *less* can reduce negative effects, but not always improves respective metrics. In addition, websites *can* benefit from different object types to be pushed. Overall, we do not find an automatically generated *one-fits-all* strategy.

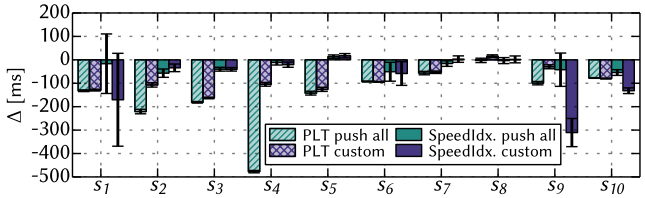


Figure 4: Custom strategies normed to the *no push* case. We show the average and the 95 % confidence interval ($\Delta < 0$ is better).

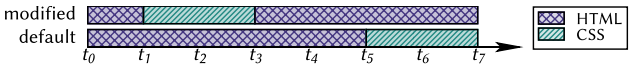
4.3 Synthetic Sites and Custom Strategies

Up to now, we assumed real-world settings, e.g., content at several servers, which can have unpredictable performance impacts. Thus, we use *synthetic* websites s_1-s_{10} that are snapshots of websites or templates, and deploy them on a *single* server, i.e., we relocate content. Again, we use request orders for *push all* and *no push* as a baseline. Instead of automatic generation (cf. Sec. 4.2), we create *custom* strategies. We inspect the browser's loading and rendering process and select resources that *i)* appear above-the-fold, or *ii)* are required to paint above-the-fold content. This can lead to benefits (cf. Fig. 4), which we discuss in two case studies. We focus on time and size improvements, as pushing less is preferable (cf. Sec. 4.2.1).

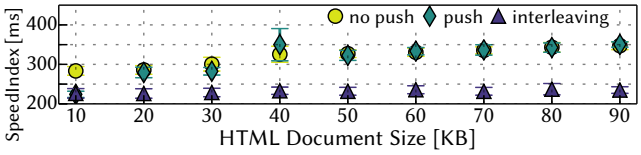
Case Studies. s_1 shows a loading icon that fades and content is shown once the DOM is ready. Thus, we push resources blocking the DOM construction (JS, CSS) and *hidden* fonts referenced in the CSS. On average, this improves SpeedIndex (but fluctuates), by only pushing 309 KB (1,057 KB, *push all*).

s_5 takes 692 ms (*push all*) vs. 1,038 ms (*no push*) to be transferred, but the metrics do not significantly improve. Regarding PLT, a blocking JS is referenced late in the <body>, which requires to create the CSSOM. This takes longer than the transfer, and the browser is not network but *computation* bound, affecting the overall process, similar to results in [32]. Our strategy pushes four render-critical resources and some images. Yet, there is no benefit, as the browser can request resources as fast as the server could push them, due to a large HTML, resulting in no network idle time. We notice the same for s_8. The HTML transfer requires multiple round trips to be completed. After the first chunk, the browser can issue requests for six render-critical resources referenced early, and using push does not change the result, similar to findings in [10].

Conclusion: Pushing all resources in a request order in this setting, i.e., *all* resources hosted on a single server, *can* reduce PLT compared to *no push*, but SpeedIndex rarely improves. Moreover, we do not observe significant detrimental effects. Yet, pushing everything can be wasteful in terms of bandwidth, e.g., if the resource is already cached, and cause contention between objects [22]. For



(a) h2o's default scheduler treats a push (e.g., CSS) as child of the parent stream (HTML). If the parent does not block, the *entire* stream is sent, possibly delaying critical resources. Our modification stops after a defined offset to start pushing.



(b) SpeedIndex (test website) for different strategies. Our *interleaving* strategy yields a stable time (average and std. dev.).

Figure 5: Interleaving Push concept and performance example.

some websites, our custom strategy performs equally to *push all* by pushing *fewer* resources. Still, even by manual inspection of the page load process, we are unable to optimize the SpeedIndex significantly for many websites. We thus conclude that the *optimal* push strategy is *highly* website-specific and requires manual effort, an in-depth understanding of the page load and render process, as well as the interplay of resources.

5 INTERLEAVING PUSH

We saw that the benefits of push depend on the size of the base document and the position of resource references, e.g., pushing objects referenced late in large base documents can be beneficial while pushing early referenced objects may not. We also observed that the order of pushes is performance critical [10] (cf. Sec. 4.2). Thus, our goal is to analyze if *interleaving* the base document with pushed objects can be beneficial. The intuition is to push the *right resources* at the *right time*.

Motivating Example. We create a website that references CSS in the <head> section and vary the size of the <body> by adding text. As baseline, the *i)* browser requests the CSS *(no push)*. Next, we *ii) push* the CSS upon request for the HTML. Last, we *iii) interleave* the delivery of HTML such that after a fixed offset, the server makes a *hard* switch to push the CSS, before proceeding with the HTML. Thereby, we incorporate page-specific knowledge into the strategy.

For the latter, we modify h2o's stream scheduler. Per default, a push is treated as child of the parent stream, e.g., a CSS as child of the `index.html`. Thus, the server pushes if the parent stream blocks, e.g., due to extra fetches if not present at the server, or is finished. Our *modification* stops the parent stream after a *defined* offset, e.g., after </head> and first bytes of <body>, and starts to push (cf. Fig. 5(a)). In the *no push* case, Chromium assigns a lower priority to CSS than for the HTML. The h2o server adheres to this and sends the CSS *after* the HTML. Here, *no push* and *push* perform similar (cf. Fig. 5(b)), as the parent does not block. *Interleaving* via push yields nearly constant and faster performance.

Real-World Websites. Motivated by this potential, we now focus on the applicability on real websites. Our set w_1-w_{20} (cf. Tab. 1) covers a broad range of content *and* website structures, e.g., w_5 consists of 8 requests served by one server, while w_{17} consists of 369 requests to 81 servers. Given this complexity and our prior analysis of browser internals, we perform measurements with *distinct*

w_1	wikipedia*	w_6	chase	w_{11}	aliexpress	w_{16}	twitter†
w_2	apple	w_7	reddit	w_{12}	ebay	w_{17}	cnn
w_3	yahoo	w_8	bestbuy	w_{13}	yelp	w_{18}	wellsfargo
w_4	amazon	w_9	paypal	w_{14}	youtube	w_{19}	bankofamerica
w_5	craigslist	w_{10}	walmart	w_{15}	microsoft	w_{20}	nytimes

Table 1: Websites (.com) selected for *interleaving push*. If not marked, we capture the landing page. (*Article, †Profile.)

modifications. We unify domains of the same infrastructure, e.g., `img.bbystatic.com` and `bestbuy.com`, and, based on inspection of the rendering process, host critical above-the-fold resources.

Strategies. As the baseline, we *i)* use a *no push* strategy. We extend this to *ii)* a *no push optimized* strategy, where we use `penthouse` [4] to compute a *critical CSS* from the included CSS, that is required to display above-the-fold content, inspired by [10, 14]. We reference this critical CSS in the <head> section and all *other* CSS at the end of the <body>. In the *iii) push all* strategy, we push *all* resources hosted on the previously merged domains, which might include additional non-critical resources. We extend this setting to the *iv) push all optimized* strategy. Here, we first push the critical CSS and critical above-the-fold resources in an interleaved fashion, and after the HTML, all other pushable resources. In the *v) push critical* strategy, we push *only* critical resources for above-the-fold content. Last, *vi)* the *push critical optimized* strategy adds the critical CSS modification to the prior strategy. For all *optimized* strategies, we use the *modified* server and the *default* in all other cases.

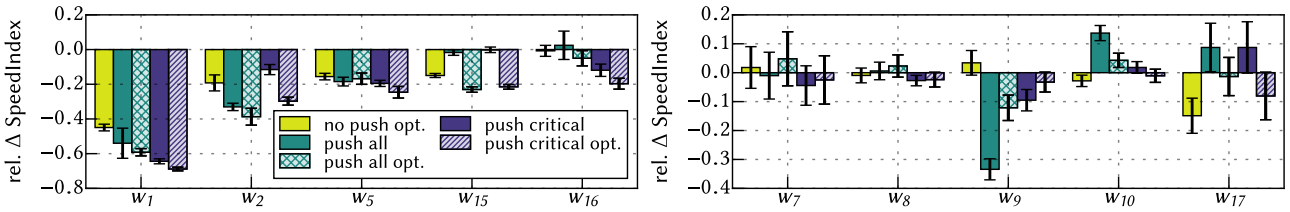
Evaluation. Using the *push critical optimized* strategy, we see benefits for five websites (cf. Fig. 6(a)), but impairments or no advance (< 10 %) for all others. Next, we focus on a representative set (cf. Fig. 6(b)), discuss influence factors, and summarize[2]. Reported changes are averages and sizes are obtained on the protocol level.

The SpeedIndex of w_1 is reduced by 44.95 % using only a critical CSS (*no push optimized*). With the *push all optimized* strategy, we see an improvement of 59.19 %, and even 68.85 % for *push critical optimized*. In the latter, we push 78.43 KB compared to 1,123 KB, saving 93 %. Here, interleaving push is beneficial, because of a large HTML size (236 KB compressed). In the *no push* case, the browser prioritizes the HTML over the CSS, and thus the server first sends the *entire* HTML. In our case, we push critical CSS after 4 KB of HTML, enabling to construct the DOM faster, and also push a blocking JS and two images, before continuing with the HTML.

w_2 already shows an improvement of 19.22 % when using a critical CSS in comparison to the *no push* strategy. In combination with the *push all* and *push critical* strategy, this yields the best performance, i.e., improvements of 33.05 % and 38.7 %. Our *push critical optimized* strategy achieves a competitive improvement of 29.74 %, only pushing 289.63 KB instead of 725.57 KB compared to the *push all optimized* strategy, saving ~ 60 %. In the default case, several CSS requested *after* the HTML block the execution of JS and thus the DOM construction. With interleaving push, we extract the critical CSS and thus reduce the critical render path.

For w_{16}, creating a critical CSS is not beneficial, as the website already uses such optimizations. Still, using our *push critical optimized* strategy improves performance by 19.67 %, pushing 10.2 KB of resources. w_{16} has a similar setting as w_1, i.e., CSS is made dependent on the HTML (45 KB compressed). With interleaving push,

[2]More results available at https://push.netray.io



(a) Websites with improvements (≥ 20 %) for our *push critical optimized* strategy over the *no push* case.

(b) Exemplary websites with detrimental effects or no benefits (< 10 %) for *push critical optimized* vs. *no push*.

Figure 6: Performance of strategies (websites as in Tab. 1). We show avg. relative changes (Δ < 0 is better) and 99.5 % confidence.

the server starts pushing CSS after 12 KB of HTML, allowing to proceed with the construction of the DOM faster.

However, other websites exhibit no major improvements or even detrimental effects. Though we are able to remove 87 KB from the critical render path of website w_7 by pushing the critical CSS instead of all CSS, the overall visual progress is not affected as much, because w_7 contains a large blocking JS in the <head>. Similar effects are observed for w_8.

w_9 benefits from pushing all resources. Still, a critical CSS does not yield drastic improvements, as the HTML contains no blocking code until the end, i.e., no delay of processing.

For w_{10}, we see detrimental effects by pushing all resources, as the page contains a lot of images, which causes bandwidth contention with other push streams. Pushing only critical resources reduces detrimental effects, but does not improve over *no push*. We find that a large portion of JS is inlined into the HTML. Therefore, interleaving push is not as efficient.

w_{17} benefits from a critical CSS in the *no push optimized* case, improving by 14.88 %, but using push does not yield improvements > 8 %. By manual inspection, we see that pushing improves the time of the *first* visual change, but not the SpeedIndex, which we attribute to the structural complexity and amount of requests.

Summary. We evaluated a novel way to utilize Server Push. By interleaving the HTML with critical CSS and critical above-the-fold resources on the H2 frame level, we can improve some websites in our testbed. Still, we also see that the benefits *highly* depend on the underlying website's structure and have to be evaluated individually. This requires a *deep* understanding of the page load and rendering process in the browser. Most promising examples include websites where we find critical blocking resources, i.e., CSS or JS affecting the DOM construction, referenced early. We also observe that switching to pushing critical resources while the browser processes inlined JS can also be beneficial, similar to [32].

Still, many websites do not benefit from our optimizations in our testbed, based on various reasons. Some websites already employ optimizations such as inlining critical JS or CSS, such that a browser is not blocked after receiving the first bytes of HTML, limiting the effect of interleaving push. Also, we see that if a website contains a lot of third-party resources, e.g., w_{17}, the effects of interleaving push dilute due to the complexity of the entire page load process.

6 DISCUSSION

We observed that the *optimal* push strategy is *highly* website specific and requires in-depth analysis of the page load process. In the following, based on our findings in Sec. 5, we discuss how a CDN *could* employ our testbed and interleaving push approach as one possibility to generate strategies automatically.

Use in CDN Deployments. Some CDNs already employ *Real User Measurements* (RUM) to obtain browser feedback, i.e., embedded JavaScripts report the data obtained in the client's browser (e.g., resource timings or rendering events) back to the CDN for further analytics [26]. Based on information about critical resources and rendering, several (interleaving) push strategies for different versions of a website and network settings, e.g., mobile, desktop, cable or cellular, could be analyzed in our testbed. Subsequently, the performance of these strategies could be assessed in A/B tests in a real deployment against the original version [19, 21, 23, 26]. By incorporating this feedback, we believe it could be possible to learn website and browser-specific push strategies from browser interactions with CDN edge nodes. However, testing the feasibility of this approach is beyond the scope of this paper.

7 CONCLUSION

This paper investigates if the current Web is ready for Server Push. To systematically answer this question, we create an H2 testbed based on Mahimahi, which we open sourced [9], to replay *any* website in a controlled manner and subject to *any* Server Push strategy. We thoroughly study the influence of various Server Push strategies, i.e., *automatically* generated or based on *guidelines*, on two major performance metrics, i.e., PLT and SpeedIndex, for both a set of real-world and synthetic websites. Our results indicate that a recommended strategy to push all embedded objects can optimize the performance of some sites, but also decrease the performance for others. By further varying the amount, order, and type of pushed objects, we again observe benefits and detrimental effects as well, highlighting the fundamental challenge of optimal Server Push usage. By *tailoring* custom strategies and using a novel resource scheduler, we show that the performance of some popular sites can indeed be improved, with minor modifications to the deployment.

We find that, while the Web may be technically ready to support Server Push, it is no feature that can be utilized easily. If and how Server Push should be used is subject to a number of *website-specific* aspects. Non-site specific adoption can very easily lower the web performance. Thus, no general guidelines can be provided for optimal push usage, making the feature not straightforward to apply. The question here is not if the Web is ready for Server Push but if the web engineers are eager to manual tuning.

ACKNOWLEDGEMENTS

We thank our shepherd Ramesh Sitaraman and the anonymous reviewers for their insightful comments and suggestions. This work has been funded by the DFG as part of the CRC 1053 MAKI.

REFERENCES

[1] `browsertime`. https://github.com/sitespeedio/browsertime. Online 06/18/2017.
[2] `h2o`. https://h2o.examp1e.net. Online 06/18/2017.
[3] `mitmproxy`. https://mitmproxy.org/. Online 06/18/2017.
[4] `penthouse`. https://github.com/pocketjoso/penthouse. Online 06/18/2017.
[5] `SpeedIndex`. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index. Online 06/18/2017.
[6] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. 2012. Anatomy of a Large European IXP. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 163–174.
[7] Jake Archibald. HTTP/2 push is tougher than I thought. https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/. Online 06/18/2017.
[8] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor. http://www.rfc-editor.org/rfc/rfc7540.txt
[9] Benedikt Wolters and Torsten Zimmermann. 2018. Testbed Source and Measurement Results. https://github.com/COMSYS/http2-conext-push.
[10] Tom Bergan, Simon Pelchat, and Michael Buettner. Rules of Thumb for HTTP/2 Push. https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0. Online 06/18/2017.
[11] Enrico Bocchi, Luca De Cicco, Marco Mellia, and Dario Rossi. 2017. *The Web, the Users, and the MOS: Influence of HTTP/2 on User Experience*. Springer International Publishing, Cham, 47–59.
[12] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho. 2009. Seven Years and One Day: Sketching the Evolution of Internet Traffic. In *IEEE INFOCOM 2009*. 711–719.
[13] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 313–328.
[14] 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 *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 439–453.
[15] H. de Saxcé, I. Oprescu, and Y. Chen. 2015. Is HTTP/2 really faster than HTTP/1.1?. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 293–299.
[16] Utkarsh Goel, Moritz Steiner, Mike P. Wittie, Martin Flack, and Stephen Ludin. 2017. Measuring What is Not Ours: A Tale of 3rd Party Performance. In *Passive and Active Measurement*, Mohamed Ali Kaafar, Steve Uhlig, and Johanna Amann (Eds.). Springer International Publishing, Cham, 142–155.
[17] Ilya Grigorik. 2013. *High Performance Browser Networking*. O'Reilly.
[18] Ilya Grigorik and Y. Weiss. Preload. https://www.w3.org/TR/preload/. Online 06/18/2017.
[19] Remy Guercio. Test New Features and Iterate Quickly with Cloudflare Workers. https://blog.cloudflare.com/iterate-quickly-with-cloudflare-workers/. Online 10/03/2018.
[20] Bo Han, Shuai Hao, and Feng Qian. 2015. MetaPush: Cellular-Friendly Server Push For HTTP/2. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (AllThingsCellular '15)*. ACM, New York, NY, USA, 57–62.
[21] Chris Jackel. A/B testing at the edge. https://www.fastly.com/blog/ab-testing-edge. Online 10/03/2018.
[22] 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 17)*. USENIX Association, Boston, MA, 545–559.
[23] Brad Lassey. Chrome's view on Push. https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/chrome_push.pdf. Online 10/02/2018.
[24] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. 2009. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement (IMC '09)*. ACM, New York, NY, USA, 90–102.
[25] Patrick Meenan. 2013. How Fast is Your Website? *Commun. ACM* 56, 4 (April 2013), 49–55.
[26] Aman Nanner. H2 Server Push Performance. https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/akamai-server-push.pdf. Online 10/02/2018.
[27] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 417–429.
[28] Mark Nottingham. httpwg: Issue #579. https://github.com/httpwg/http-extensions/issues/579. Online 06/18/2017.
[29] Kazuho Oku and Mark Nottingham. 2017. *Cache Digests for HTTP/2*. Internet-Draft draft-ietf-httpbis-cache-digest-02. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-digest-02.txt
[30] R. Peon and H. Ruellan. 2015. *HPACK: Header Compression for HTTP/2*. RFC 7541. RFC Editor. http://www.rfc-editor.org/rfc/rfc7541.txt
[31] 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 (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 459–468.
[32] 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 (SIGCOMM '17)*. ACM, New York, NY, USA, 390–403.
[33] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. 2018. A First Look at QUIC in the Wild. In *Passive and Active Measurement*, Robert Beverly, Georgios Smaragdakis, and Anja Feldmann (Eds.). Springer International Publishing, Cham, 255–268.
[34] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. 2018. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *Proceedings of the 2018 Internet Measurement Conference (IMC '18)*. ACM, New York, NY, USA.
[35] 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.
[36] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 473–485.
[37] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How Speedy is SPDY?. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 387–399.
[38] Kyriakos Zarifis, Mark Holland, Manish Jain, Ethan Katz-Bassett, and Ramesh Govindan. 2017. *Making Effective Use of HTTP/2 Server Push in Content Delivery Networks*. Technical Report. University of Southern California.
[39] Torsten Zimmermann, Jan Rüth, Benedikt Wolters, and Oliver Hohlfeld. 2017. How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*.
[40] 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 (Internet QoE '17)*. ACM, New York, NY, USA, 1–6.