xPrio: Crosslayer Web Resource Prioritization at Runtime

CONSTANTIN SANDER, RWTH Aachen University, Germany
IKE KUNZE, CUJO AI, Norway
HENDRIK BUSCHBAUM, RWTH Aachen University, Germany
KLAUS WEHRLE, RWTH Aachen University, Germany

Speeding up webpage loads is a crosslayer optimization problem that depends on webpage structure and network conditions. Yet, traditional HTTP resource prioritization forgoes combining resource dependency and network state data, resulting in varying performance. More sophisticated optimization approaches increasingly incorporate crosslayer data, but they usually gather it a-priori, questioning the practical applicability.

We present xPrio, a reinforcement learning-based resource prioritization approach that provides a scalable middleground: it avoids costly a-priori knowledge, but still includes crosslayer data from browser and transport layer signals collected at runtime. xPrio turns this readily available information into actionable resource priorities that avoid detriments of traditional strategies and achieves mean SpeedIndex speedups above 15 % on pages of the Alexa Top 500. As such, xPrio can widely improve performance with little overhead in use.

CCS Concepts: • **Networks** → *Cross-layer protocols*; **Layering**; **Application layer protocols**; **Network experimentation**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: Crosslayer Optimization; HTTP; Prioritization; Reinforcement Learning; Web Simulation

ACM Reference Format:
Constantin Sander, Ike Kunze, Hendrik Buschbaum, and Klaus Wehrle. 2026. xPrio: Crosslayer Web Resource Prioritization at Runtime. *Proc. ACM Netw.* 4, CoNEXT2, Article 27 (June 2026), 16 pages. https://doi.org/10.1145/3808675

1 Introduction

Today's webpages are complex and consist of many resources [7] with highly diverse impact on performance [47, 48]. Thus, correctly scheduling resource transfers is crucial to avoid delays. One common approach is HTTP Resource Prioritization [33], where browsers signal resource importance to webservers to, e.g., prioritize render-blocking resources or images in the view port. The prioritization strategies are hand-crafted and differ between browsers with each strategy showing benefits and detriments in certain scenarios [26, 42, 48]. These differences can be traced back to only partially leveraged resource dependencies [42, 48] and ignored network states [42].

Addressing this weakness, various related works reschedule resource downloads based on resource dependency [11, 20–23, 31, 36, 41, 49] and network state information [11, 20, 41, 49]. Yet, these approaches use this information in an *a-priori* fashion, so expect it to be available upfront. For instance, some approaches [20, 49] rely on the exact image and script names contained on a page and their impact on performance to be known before scheduling subsequent requests. This also implies that the information needs to be precise. A differently named image or a changed impact on performance can result in ambiguously wrong decisions, prioritizing or deprioritizing the wrong

Authors' Contact Information: Constantin Sander, RWTH Aachen University, Aachen, Germany, sander@comsys.rwth-aachen.de; Ike Kunze, CUJO AI, Oslo, Norway, ike.kunze@cujo.com; Hendrik Buschbaum, RWTH Aachen University, Aachen, Germany, buschbaum@comsys.rwth-aachen.de; Klaus Wehrle, RWTH Aachen University, Aachen, Germany, wehrle@comsys.rwth-aachen.de.

resources. Thus, accurate a-priori resource dependency information is required. However, gaining accurate a-priori resource dependencies can be costly or even infeasible because extracting timely resource dependencies requires a full webpage load with every content change. As many pages are dynamically generated (cf. Appendix A.1), reevaluation prior to every actual request would be necessary, even if the underlying resources dependencies did not change. Similarly, collecting network information a-priori is challenging. While latency can be measured at connection start, loss and bandwidth estimates require longer communication or prior experience that is quickly outdated due to changing congestion, network paths, or signal quality [30, 34, 43]. Hence, applying approaches relying on a-priori information in practice is often prohibitive. However, parts of the information can be gained later *at runtime*: network states can be extracted from transport protocols, and (partial) dependency data can be deduced from browser signals. Despite this potential, research has not yet found ways to utilize this data at runtime for prioritization.

In this paper, we, thus, propose xPrio, an HTTP resource prioritization approach that leverages runtime information to find better priorities without relying on a-priori data. Specifically, xPrio combines the readily available browser prioritization signals that contain (partial) dependency information with bandwidth, RTT, and loss information from the underlying transport to guide its decisions. To integrate this multi-dimensional data, containing a wide variety of data types as well as correlations, and to derive sensible actions, xPrio uses deep reinforcement learning trained on a novel crosslayer webperformance simulation. xPrio can then make use of the userspace nature of QUIC and HTTP/3 to extract crosslayer data at runtime and adapt HTTP/3 priorities on the go.

In the following, we present and evaluate xPrio, overall making the following key contributions:

- We explain how runtime inputs from browsers and the transport layer in combination with reinforcement learning can enable finding better resource priorities
- We demonstrate how we can train xPrio with a novel crosslayer network and browser simulator and how we can incorporate these simulation results in the reinforcement learning process
- We evaluate the performance of xPrio, show that it can achieve average improvements above 15 % on the SpeedIndex and industry metrics, and analyze the learned strategy.

Our simulation, training and evaluation implementation is available on GitHub [40] (cf. A.4).

2 Background and Motivation

Loading a webpage consists of multiple intertwined processes [2, 47]. First, the webpage's HTML document is downloaded and parsed into a Document Object Model (DOM). New resources are hereby discovered and subsequently also downloaded. Concurrently, the parsing continues and the DOM is, eventually, rendered and painted to screen. Finally, objects are positioned at the right positions and can be (progressively) shown when their underlying resource is received.

Blocking Resources and Critical Path. Certain resources, such as JavaScripts or stylesheets, can interact with the DOM or its visual appearance. To avoid inconsistent states, synchronous scripts block parsing and rendering until they are fully loaded and evaluated. Similarly, stylesheets block rendering. Together, these blocking dependencies create a *critical path* [47] that has to be processed step-by-step until the page is fully parsed / rendered. Prioritizing the critical path can reduce blocking and speed up the page load.

Performance Metrics and Prioritization. To characterize a page load, the Page Load Time (PLT) can be used to measure the time until all resources have been loaded. However, it correlates poorly with user-perceived performance [18, 37, 50] as it ignores that certain resources, such as above-the-fold (ATF) images directly visible on screen, are visually more important than others. The SpeedIndex (SI), thus, captures the visual progress of a webpage and research finds that it indeed correlates strongly with user experience [18, 37, 50]. As such, user-focused page loading

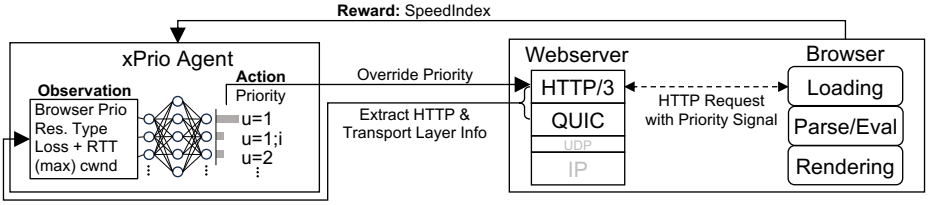


Fig. 1. xPrio's Design: Reinforcement Learning for crosslayer prioritization optimization. Crosslayer information is fed as input (observation) into a neural network to decide on the prioritization (action) of resources with the goal to improve the SpeedIndex (reward). Importantly, no external, a-priori knowledge is introduced.

needs to also prioritize, e.g., ATF resources for perceivable speedups. In order to implement such a fine prioritization, HTTP introduces Resource Prioritization to announce resource importance.

2.1 HTTP Resource Prioritization

Resource Prioritization allows browsers to signal webservers a desired transmission order for requested resources [9, 27, 33, 48]. Its current form — the extensible prioritization scheme (EPS) [33] — assigns one of eight *urgencies* (u) to every resource and an *incremental* (i) flag. Resources are then sent in increasing order of urgency, where for every urgency first non-incremental resources are sent in sequential order and incremental resources are sent, subsequently, in round-robin.

Prioritization Strategies. Browsers use different strategies for determining which priorities to signal. Chrome, e.g., uses a sequential approach to specifically prioritize critical-path and ATF resources while Firefox used a complex, group-based and weighted schedule in earlier HTTP versions [48]. Yet, as the EPS cannot represent these group weights anymore, Firefox nowadays also adopts a sequential approach, while Chrome recently tested incremental schedules for images [17].

The Quest for Performance. Research shows that the performance of the strategies depends on website structure and network state [26, 27, 42, 48]. For example, Chrome's sequential loading often provides best performance, but there are cases where round-robin excels, especially in challenging network scenarios [42]. As a result, related work combines dependency and network information to precompute better schedules [41, 49]. However, dynamic changes in network state or website resource dependencies challenge this precomputation. E.g., dynamic or personalized content (cf. Appendix A.1) can incur fluctuating resource dependencies such that frequent reevaluations, i.e., costly full webpage loads, are required to even check for changes. Network states, such as bandwidth estimates, require longer data transmission, so are not available upfront. While historic data could be used instead, it is volatile due to network path changes or fluctuating wireless access [30, 34, 43]. As such, gathering accurate a-priori information is either expensive or infeasible.

However, HTTP resource prioritization does not actually need a precomputed schedule as it prioritizes requests as they arrive. Thus, network and dependency information could be extracted from the transport or HTTP layer at runtime. Yet, this angle for improving prioritization has not yet been studied before, which is why we set out to leverage it in xPrio's design as we detail next.

3 Design Overview

The main goal of xPrio is to combine network and browser information for dynamic resource prioritization at runtime. To do so, it directly interfaces with an HTTP/3 webserver, as illustrated in Fig. 1. For every request, xPrio intercepts the prioritization signal from the browser and taps into the underlying transport layer to extract its network state. Based on the transport and the browser information, it then rewrites the prioritization and the actual request processing continues.

Accessing Prioritization and Crosslayer Information. To access browser information, we extract resource requests and their prioritization on the HTTP layer. For obtaining crosslayer information, we leverage that QUIC, the default transport of HTTP/3, also resides in userspace [10, 19]. Thus, with every HTTP/3 request, we can easily access the full transport connection state. Specifically, we extract loss rate, RTT, and congestion window (cwnd) estimates from QUIC's retransmission and congestion control logic.

Taming the Parameter Space. Combining standard browser resource information (priority, resource type) with continuous network state information into one input vector spans a complex parameter space that needs to be thoroughly analyzed. In particular, there is a wide variety of patterns and correlations with subtle impacts, e.g., round robin scheduling finding a break even with growing loss on certain websites, that need to be taken into account [26, 42]. Analyzing these correlations by hand requires detailed testing of all possible variations, including *per-webpage* resource dependency checks and performance derivations that need to be avoided for generalized application. For every resource, updated crosslayer information could make a different scheduling decision necessary. Thus, similar to related work [20, 25, 49], we equip xPrio with an automated approach using deep reinforcement learning (DRL) [29] to explore the parameter space and exploit previous experience for finding sensible prioritization decisions without manual tuning. Specifically, our xPrio DRL *agent* observes a webpage loading *environment* through the crosslayer information. It then takes *action* by changing the EPS resource priorities to improve the perception-correlated SI metric, which serves as our *reward*. We present our DRL setup in more detail in Sec. 4.

Gaining Training Samples. DRL requires thousands of samples of the environment for training, so we follow common practice for reinforcement learning [20, 25, 46] and base our training on a simulation which does not only speed up training, but also provides reproducible and repeatable runs that result in low noise, further aiding DRL. In this work, we design a focused crosslayer web performance simulation that enables us to extract transport information of real HTTP/3 stacks while also emitting SI estimates as rewards. Notably, our simulation closely models the intricate influences of prioritization, the critical path, evaluation durations, resource dependencies, and their impact on the SI and, thus, serves as an accurate basis for our training environment. We discuss the nuances of our simulation in detail in Sec. 5.

Deploying in Real-World. Once trained, xPrio can drive real-world resource schedules by applying its learned strategies to real webservers. For this, xPrio connects to a real HTTP/3 webserver that serves the actual webpage to the user and we employ the same procedure as in Fig. 1, feeding crosslayer information into xPrio and overriding prioritization at runtime with its output. As we intentionally refrain from a-priori information, no further information exchange is necessary.

After having described the general design of xPrio, we next describe its DRL model in Sec. 4 followed by the simulation used in training in Sec. 5.

4 Deep Reinforcement Learning Model

Reinforcement learning [46] consists of an *environment* and an *agent*. The latter repeatedly gains *observations* from the environment and triggers *actions* to manipulate it during an *episode*. For every action, a *reward* is received and the objective is to maximize the *cumulated* reward (*return*) of an episode. We map this concept to resource prioritization as illustrated in Fig. 1 to accommodate the wide variety of inputs and derive sensible actions. Our environment captures the webpage loading process and entails the server and the browser that our agent interacts with; correspondingly, one page load represents one episode. Every resource request triggers an observation of the crosslayer and prioritization information, which our agent uses to compute an action that sets the resource's priority in order to maximize the return, in our case the negative SI.

4.1 Input and Output Spaces

We use the following inputs and outputs for our reinforcement learning setup [46].

Actions. Our agent chooses its actions from an action space consisting of 16 discrete actions, namely all combinations of the 8 urgencies (u) of the extensible prioritization scheme (EPS) and its incremental (i) flag. For each resource request, we ask the agent for its decision on the new resource priority. Internally, our agent holds a representation of the expected return for every decision (cf. Sec. 4.2) and selects the action that maximizes the expected return.

Observations. Our agent bases its decisions on various input features, which we gather for each request. Our features consist of the input prioritization signal, the resource's class derived from the `sec-fetch-dest` request header, the number of previously requested resources of the same class, the current/maximum cwnd, smoothed/minimum RTT and packet loss. These features inform the agent about the browser's view, specific resource information, and crosslayer transport information.

Reward. The agent receives rewards for its actions and aims to maximize the cumulative reward / return of an episode. Hence, our reward substantially influences the behavior of the agent and we choose the SI as the underlying metric due to its strong correlation with human perception of performance [18, 37, 50]. Yet, as lower SI values correspond to faster page loads, we use the negative SI, to enforce minimization, and scale it against Chrome's prioritization strategy to normalize it for different webpages. Formally, our reward for one webpage and the t^{th} action is defined as follows:

$$r_t^\pi = \begin{cases} -\frac{SI^\pi - SI^{base}}{SI^{base}}, & \text{if } t = T, \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where SI^π is the SI gained with our trained policy (π) and SI^{base} is the Chrome baseline SI. Experiments splitting the SI into parts for faster convergence [46] did not show benefits.

4.2 Implementation

Our Agent is implemented as a Deep-Q-Network (DQN) [29], contrasting popular actor-critic methods [28, 44] that got stuck in local optima. The underlying neural network is a small, efficient 3-layer MLP with 64 units per layer and relu activation functions implemented in TensorFlow [16]. We train the network with 1M episodes using RMSProp, a learning rate of 0.001 and an epsilon declining by 10% every 100k episodes. For connecting ns3 [35] with our training [16], we use ns3-gym [15]. Further details can be found in our opensource implementation [40](cf. A.4).

Having discussed how xPrio uses DRL, we next describe how simulations aid its training.

5 Crosslayer Simulation

We train our DRL model via simulations to avoid infeasibly long real webperformance measurements and quickly gain environment samples. However, when studying existing web simulators [20, 47], we found two limitations that challenge training of xPrio: First, xPrio requires fine-grained crosslayer information from the transport layer, which existing simulators do not provide. Second, browser prioritization decisions and impacts are generally not modeled. To address these limitations, we introduce a novel, packet-level web simulator that 1) embeds a real QUIC stack to obtain realistic transport layer information, and 2) integrates a model of the webpage loading pipeline that accounts for changes in prioritization. In the following, we present our simulator in more detail.

5.1 QUIC-based HTTP Transport

Prioritization has intricate crosslayer interactions [42], such that xPrio uses fine-grained transport layer information for controlling its resource schedules. Aiming to accurately model these interactions, we integrate the mature *quicly* QUIC stack [4] from fastly's h2o webserver [6], which

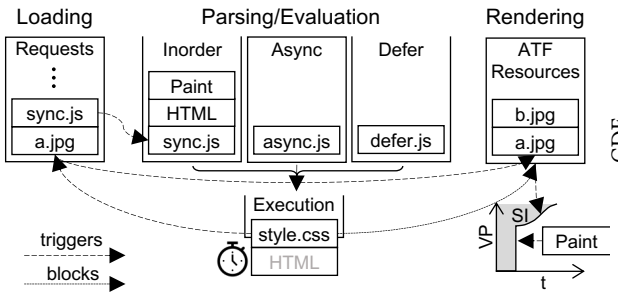


Fig. 2. xPrio's Webpage SpeedIndex Simulator

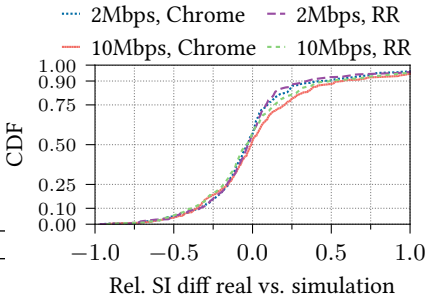


Fig. 3. SpeedIndex Simulation Accuracy

we later also use for evaluation, into the proven, packet-based network simulator ns3 [35]. This way, we can efficiently transfer data via QUIC, extract realistic congestion control information and leverage h2o's EPS implementation for prioritization. Overall, our combination enables simulating HTTP requests with correct congestion control, retransmission, and prioritization behavior.

5.2 Pageload and SpeedIndex Simulation

Besides the transport layer, our simulation also needs to correctly reflect the pageload behavior, as HTTP prioritization can, e.g., delay resources on the critical path (cf. Sec. 2) causing blocking. To accurately reflect these considerations, our simulation needs to model the browser pipeline and the impact on user-perceivable performance. To do so, we source resource and critical-path dependencies as well as the visual impact of corresponding events from real browser page loads. This information then fuels a simplified state machine to efficiently mimic the browser's behavior.

5.2.1 Real-world Browser Trace Input. For each page of our training corpus, we load it once using the Chromium browser with enabled tracing [8, 45]. We then analyze the trace to factor out network variance and obtain a cleaned dependency graph augmented with information on the browser's inner method calls specifically geared toward assessing user-perceivable impacts. In particular, we gather involved resources with their HTTP priorities and dependencies, reconstruct the critical path, and identify event sequences and the DOM elements that are impacted by these events. Subsequently, we analyze the impact of DOM elements and images on the visual appearance by hiding/showing them and taking screenshots of the page to compute its visual progress. Fusing this information, we create a schedule of resources, dependent browser events, and rendering information to represent how loading a specific resource triggers further events and impacts the SI.

5.2.2 Schedule Replay. Our simulated browser remodels Chromium's decisions and uses the generated schedule to replay events and resource requests as illustrated in Fig. 2.
Object Loading. We first request the main HTML file and then simulate a preload scanner [2] that loads further resources from the HTML without fully parsing it (Fig. 2, left).
Parsing/Evaluation. Concurrently, our simulated browser replays HTML parsing and script evaluation events. For this, we assign events to one of three queues for 1) strict in-order execution (sync. scripts, stylesheets, and HTML parsing) as well as 2) asynchronous and 3) deferred scripts. We select the next event to be executed based on the queues, check if necessary resources have been fully downloaded, and, subsequently, evaluate the corresponding resource, i.e., wait the previously traced evaluation duration and emit potential new resource requests and events.

Rendering. Browsers wait for the first paint event to then unlock the rendering pipeline. xPrio remodels this behavior by also replaying paint events and unlocking its virtual rendering pipeline. Thereafter, we increase a virtual visual progress whenever a visually impactful event is triggered, e.g., a JavaScript adding an impactful DOM element, identified in our trace schedule. The virtual visual progress can then be used to compute the SI and fuel the training of xPrio (cf. Sec. 4).

5.3 Simulation Accuracy and Runtime

The effectiveness of our training is directly tied to our simulation framework. Hence, we evaluate our simulation accuracy and runtime in dedicated experiments.

Accuracy. Focusing on accuracy, we first compare SI measurements with Chrome to our simulation. For the Chrome measurements, we use the same methodology as in our evaluation (cf. Sec. 6), measuring the Alexa Top-500 corpus with Chrome's prioritization (*Chrome*) and a round-robin schedule (*RR*). For our simulation, we first gather traces and replay accordingly.

Fig. 3 shows the relative differences between Chrome and our simulation for bandwidths of 2 and 10 Mbps and an RTT of 10 ms. The majority of websites can be simulated with less than 20 % difference in SI. Only for 20 % of the websites, we over/underestimate the SI by more than 35 %. We find two major reasons for this behavior: (1) Our simulation assumes a monotonically increasing visual progress. However, layout shifts can incur a reduction in progress and underestimated SI values. (2) Our simulation maps visual impact to the last event that adapts an element. Repeated adaptions can be wrongly mapped to later events, such that we overestimate the SI. While improved tracing can help to reduce these effects, our simulation already matches the trend of the SI.

Runtime. We also tested our simulation's runtime. Our implementation gives room for several optimizations, yet achieves a median runtime of ~0.332 s. In contrast, Chrome measurements at 2 Mbps/10ms RTT incur median runtimes of ~58.179 s, including setup and SI processing. Moreover, our simulation is highly parallelizable while Chrome can be influenced by parallelization. Hence, we can robustly gather multiple orders of magnitudes more runs via our simulation.

6 Evaluation

xPrio along with its training algorithm and simulation framework is designed for broad applicability and to achieve meaningful performance benefits by leveraging crosslayer information at runtime. In this section, we experimentally evaluate xPrio with real page loads to demonstrate its potential and to characterize in which scenarios end users can benefit most from servers using xPrio.

6.1 Experimental Setup

For our experiments, we use a testbed setup to host and load real-world websites in a controlled environment. Similar to Mahimahi [32], we first download the pages and then replay them through a virtual connection between server and client to represent different network conditions. For the replay, we host an h2o server extended with xPrio and access it using Chrome and Browsertime to measure the SI. Our whole testbed runs on consumer-grade machines with an Intel i5-4590 CPU and 16GB of RAM. During the replay, we use a single connection [38] and either passthrough Chrome's prioritization signals, use xPrio decisions, or signal RR scheduling. We additionally use an adaption of Firefox's HTTP/2 strategy from literature (cf. Sec. 2.1, [42]).

Our test corpus comprises of the landing pages of the Alexa Top 500 Toplist from May 2023. For 26 of these websites, downloading failed due to DNS or server errors and 22 websites could not be replayed without exceeding a 5 minute timeout; we exclude these pages from our corpus. We train xPrio on one half of the remaining pages and evaluate its performance using the other half.

We measure the performance of the studied approaches using the SI and present the relative difference between xPrio and the other prioritization strategies. All metrics have been measured

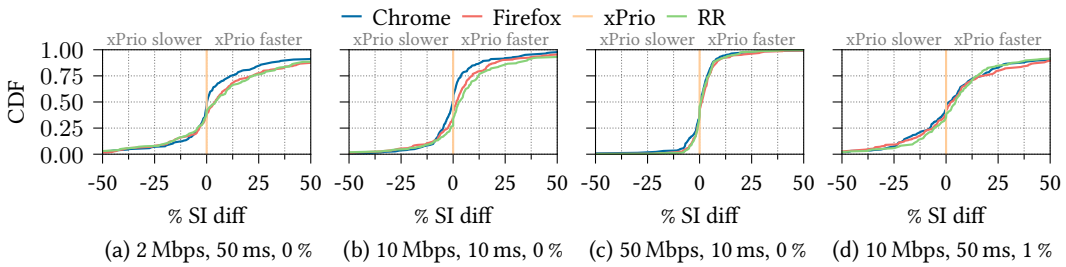


Fig. 4. Relative SpeedIndex difference between xPrio and traditional browser strategies. Values above zero correspond to cases where xPrio loads the webpage quicker, values below zero to the opposite.

with Browsertime and a full HD viewport where we ensured that websites show a steady view to avoid noise on the SI [18] by delaying image carousels or hiding cookie banners. Every measurement was repeated 5 times and we use the corresponding median in the following unless noted otherwise.

6.2 xPrio's Impact on the SpeedIndex

Fig. 4 shows xPrio's SI performance in relation to the other strategies when trained on one half of our corpus and evaluated on the other half. As can be seen, xPrio consistently outperforms the other strategies, improving performance for more websites than degrading it. For example, at 2 Mbps and 50 ms RTT, it improves the SI by more than 25 % for 15 %/23 % of the pages in comparison to Chrome/RR, while degrading it by 25 % for only 7 %/8 % of the pages. Overall, xPrio improves the SI by 18 % on average when compared to Firefox and 13 %/14 % compared to Chrome and RR. With higher RTTs, e.g. 100 ms, we see a slight reduction to 8 %-12 % (not shown). Similarly, the improvements also decrease with increasing bandwidths with xPrio achieving average improvements of 3 % to 9 % at 10 Mbps, 10 ms RTT (Fig. 4b) and 1 % to 3 % at 50 Mbps (Fig. 4c). This means that xPrio shows its highest benefits with low RTTs and low bandwidths, while higher bandwidths or RTTs show less improvements. However, when also factoring in packet loss, we find that xPrio can bring benefits irrespective of the configured RTT. For example, with 1 % loss at 10 Mbps and 10 ms RTT, mean improvements from 5 % onwards can be achieved (not shown) while we find stronger benefits for an RTT of 50 ms with mean speedups of 9 % to 13 % (cf. Fig. 4d). We conclude that xPrio can generally bring benefits in lossy settings, such as in wireless scenarios.

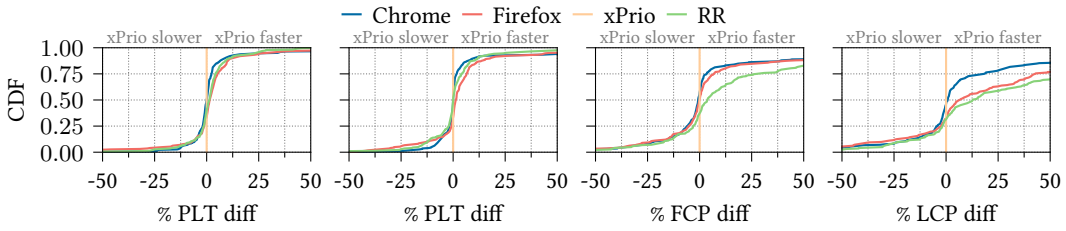
We also evaluated xPrio in a real-world wireless scenario with Starlink and compared its performance against related work; these results support our previous findings (cf. Appendix A.3).

Takeaway. *xPrio can speed up the SI by up to about* 15 % *on average in low-bandwidth or high-loss scenarios, while avoiding detriments incurred by the strategies used by Firefox and Chrome, and RR.*

With its focus on the SI, xPrio explicitly aims to optimize the user-perceived web performance. In contrast, common industry practices for web optimization rely on a mixture of additional metrics [12], such as first / largest contentful paint (FCP / LCP) or the traditional, less user-centric PLT. Hence, we next also analyze xPrio's impact on these metrics and web performance in general.

6.3 xPrio's Implicit Impact on Other Metrics

Fig. 5 shows xPrio's performance for the PLT, FCP, and LCP. For the PLT (Fig. 5a and 5b), xPrio achieves average improvements of 4 % for 10 Mbps and 10 ms RTT and 9 % for 2 Mbps and 50 ms RTT in comparison to Chrome. This means that xPrio does not only improve the perceivable performance, but also ensures that all resources are fully transferred earlier.



(a) 10 Mbps, 10 ms, 0 % (b) 2 Mbps, 50 ms, 0 % (c) 2 Mbps, 50 ms, 0 % (d) 2 Mbps, 50 ms, 0 %

Fig. 5. Impact of xPrio on standard web metrics that it does not explicitly optimize on.

Scenario	Browser Priority Signal	Resource Type	Resource Size	Network State
10 Mbps, 10 ms, 0 %	−74.0 %	−51.0 %	−17.6 %	−8.2 %
2 Mbps, 100 ms, 0 %	−37.5 %	−34.5 %	−30.1 %	−7.4 %

Table 1. Degradation of mean SpeedIndex improvements when ignoring certain features.

One of the main reasons for this speedup is that xPrio reorders resource transfers to reduce the waiting time for blocking resources which can also be seen in the improved FCP in Fig. 5c. Specifically, the FCP measures the time until first parts of the page can be shown, which requires that all blocking resources in the header of the page are fully evaluated. xPrio reduces this time by 21 % to 23 % on average at 2 Mbps and 50 ms RTT compared to Chrome and RR.

Finally, LCP measures when the largest element of a page, often an image, is shown. For this metric, xPrio achieves the largest gain, improving performance by 23 % for Chrome, 39 % for Firefox, and 50 % for RR, highlighting the benefits of the visual guidance provided by the SI.

Takeaway. *xPrio reaches average improvements above* 20 % *for paint metrics and up to* 9 % *for the PLT, highlighting the positive impact of xPrio beyond the SpeedIndex for other standard web metrics.*

Having shown the benefits of xPrio, we finally aim to deduce how it achieves these improvements.

6.4 xPrio's Strategy and Decisions

To analyze the prioritization strategy of xPrio, we first study the impact of different input features on its performance in an ablation study before dissecting its behavior to explain its decisions.

6.4.1 Ablation Study. For our ablation study, we train subvariants of xPrio that leave out certain features. We then evaluate these subvariants in our simulator and compare their performance.

Table 1 shows the performance degradation at 10 Mbps/10 ms and 2 Mbps/100 ms. The biggest impact (74.0 % at 10Mbps, 37.5 % at 2Mbps) can be seen when ignoring the browser's priority signal, showing that xPrio reuses the dependency information encoded therein. Similarly important (51.0 % / 34.5 %) is the resource type that allows for further differentiation of the priority signal. For both signals, we can see that the impact shrinks for lower bandwidths, while the impact of the network state stays equal at about 8 % and the impact of the resource size grows (17.6 % to 30.1 %). We derive that the network state has a smaller but steady impact on performance. Contrasting, the resource size is critical in low-bandwidth scenarios, as it allows to estimate resource transfer times, while the resource processing characteristics are more important in high-bandwidth scenarios.

6.4.2 Strategy Analysis. Inspecting xPrio's learned strategy in more detail, we extract its behavior by learning decision trees on our previous simulated runs. While we find that the decisions depend on a multitude of inputs, the most general impacts can be summarized as follows.

Sequential vs. Incremental. JavaScript and images are mostly sent sequentially. Notably, the latter choice stands in contrast to recent Chrome experiments [17]. The remaining resources are sent incrementally for larger bandwidths, higher loss, or big HTML resources, i.e., when bandwidth is not limiting processing, HoL Blocking [42] can be avoided or resources are parsed incrementally.
Urgency. Urgency strongly depends on resource criticality and request order. Criticial JavaScript is prioritized very high, while CSS is prioritized only high; we attribute this to CSS not explicitly blocking parsing. Early image requests get a high priority, while later requests are deprioritized. We conjecture that xPrio identifies ATF images by being sent early on. Last, we find that smaller resources incur higher priorities, as they incur a lower risk of wasting bandwidth.
Takeaway. *In total, xPrio reuses detailed resource information for governing the urgency of requests, while the incremental flag is more broadly set depending on general information and network state.*

7 Related Work

In the realm of web performance acceleration, there exist several lines of related research that optimize web resource transfer schedules and the order that resources are requested.
Client-Side Request Scheduling. One line of research discusses the client-based scheduling of resources. SipLoader [23] infers dependencies between resources and the SI to then request the visually most important resources via JavaScript first. Polaris [31] analyzes the dependencies between JavaScript resources and then schedules and evaluates them explicitly while skipping blocking/waiting on JavaScript if not necessary. Both approaches require a-priori data and alter webpages to implement their schedulers. This altering can be error-prone, impacting the appearance of a page [41]. xPrio is transparent to the webpage content and does not require a-priori data.
HTTP Prioritization. Various works analyze the influence of browser strategies for HTTP prioritization [27, 42, 48] or propose new strategies [27]. They find two main results: First, no single, manually tweaked strategy is best, and, second, crosslayer interdependencies can impact performance. xPrio thus uses an automated, data-driven approach and furthermore includes crosslayer information. While VGprio [41] and DRP-RL [49] also present data-driven approaches (even using DRL) to crosslayer HTTP prioritization, both use a-priori dependency and network information and compute their schedules upfront. When these signals are up to date, greater improvements than with xPrio can be achieved, since, e.g., post-load knowledge about the page can be leveraged. Yet, outdated data for, e.g., resources can lead to mismatches such that resources are wrongly (de-)prioritized and performance is degraded. xPrio only uses runtime data that is recent by design, but can contain uncertainty due to a partial view on the page and the network. However, because this uncertainty is already part of the training on multiple pages, the learning process can account for it. As such, xPrio will not achieve the best case results of a-priori approaches, but page variations do not affect recency or uncertainty, resulting in xPrio's improvements with unseen pages.
Server Push. Another line of research investigates HTTP server push to preemptively schedule resource transfers. Vroom [36], HTTP Steady Connections [22] and Klotski [11] use dependency information to push critical resources. Alohamora [20] extends this data with a-priori network and client data processed via DRL. Webgaze [21] uses eye-tracking to push resources that users look at.

All approaches require exact a-priori data, such as dependency information, and partly even further costly or error-prone data. Additionally, Server Push has been deprecated by Chromium [1] and Firefox [3] such that >80% of web users [5] cannot use it. xPrio, again, requires no detailed a-priori information and solely bases on HTTP prioritization with active development [33].

8 Discussion

After evaluating xPrio and putting it into context with related work, we now discuss our findings with a focus on further application cases of xPrio and its performance in the long run.

Overhead and Client-based Deployment. xPrio's overhead is split into training and inference overhead. Training our (non-optimized) prototype with 32 simulation threads on two Intel Xeon 8160 CPUs takes ∼9 hours. However, we argue that training time is negligible as it is only required once. In contrast, inference overhead occurs per request on server-side. Yet, xPrio relies on a light-weight 3-layer neural network, which we deem small and incurs low overhead. Alternatively, xPrio could also be deployed on client-side. Most of the inputs are known by the browser, only the server cwnd needs to be replaced with client bandwidth estimates and resource sizes need to be read from HTTP response headers, where then reprioritization [33] after receiving the headers is necessary. Hence, running xPrio on browsers should be feasible removing server-side overhead.
Other Metrics/Browsers/Network Stacks. xPrio currently uses the SI as its objective and only implicitly improves other web metrics (cf. Sec. 6.3). Yet, xPrio can also explicitly optimize other metrics, as long as they can be simulated; we then expect even stronger improvements.

Moreover, xPrio was designed for HTTP/3 and Chromium, but can be extended beyond. While our simulation was tailored for Chromium, adaption to Firefox should be possible. With significant effort, we also think that HTTP/2 support is possible by instrumenting the Linux / ns-3 TCP stack.
Bandwidth and Prioritization. We see that xPrio and prioritization in general is less effective in high-bandwidth scenarios, where network bottlenecks vanish and client-side processing influences performance most. As such, with ever-increasing bandwidths, it can be argued that xPrio will get less effective. Yet, historically, a steady increase in webpage size and resource counts can be seen [7], increasing weight on the network. Additionally, modern CPUs are getting more and more powerful, shifting the bottleneck further from processing to networking. Thus, we believe that also future developments in bandwidth and page weight will make xPrio a valuable tool for improvements. Moreover, a steady coverage of broadband connections is not guaranteed: bandwidths above 10 Mbps are still not ubiquitous, e.g., in suburban areas [24], and challenging mobile environments can also present much lower bandwidths at every point in time – also with 5G [34]. Hence, the conceptual contributions of xPrio are not only beneficial on a short-term basis, but also long-term.
Generalization vs. Specialization. Our xPrio design is meant to generalize, yet xPrio can also be specialized. For instance, we saw more than twice the SI improvements when training for specific pages, in which case the performance, of course, did not generalize. As such, we argue that for specific webpage use-cases where general structure is not changed frequently, xPrio can be fine-tuned. In this case, it still does not need a-priori data, but we see the fine-tuning as a form of special input, such that we deem it as a middle ground between xPrio and a-priori approaches.

9 Conclusion

In this paper, we presented xPrio, a crosslayer-guided HTTP resource prioritization approach that avoids costly a-priori resource and network information, but only relies on runtime data. Specifically, xPrio reuses transport and HTTP layer information and combines these via Deep Reinforcement Learning trained on crosslayer webperformance simulations to learn how to prioritize resources more effectively. xPrio achieves average SI speedups of up to 18 % in comparison to traditional prioritization methods in bandwidth-limited scenarios, where several pages see improvements above 25 % and only few see similar detriments. Additionally, we find that the benefits also manifest in real-world scenarios and on metrics beyond the SI. As such, xPrio presents a valuable tool for practically applicable, crosslayer-guided prioritization that many webpages can benefit from.

Acknowledgments

This work was funded by the German Research Foundation DFG under Grant No. WE 2935/20-1 (LEGATO). We thank Robin Marx, the anonymous reviewers and our shepherd Dirk Kutscher for the fruitful discussions and their valuable feedback.

References

[1] 2022. Removing HTTP/2 Server Push from Chrome. https://developer.chrome.com/blog/removing-push/. (Accessed on 12/12/2025).

[2] 2022. What's a preload scanner. https://web.dev/articles/preload-scanner#whats_a_preload_scanner. (Accessed on 12/12/2025).

[3] 2024. Firefox 132.0, See All New Features, Updates and Fixes. https://www.firefox.com/en-US/firefox/132.0/releasenotes. (Accessed on 12/12/2025).

[4] 2025. A modular QUIC stack designed primarily for H2O. https://github.com/h2o/quicly. (Accessed on 12/12/2025).

[5] 2025. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/#monthly-202512-202512-bar. (Accessed on 12/12/2025).

[6] 2025. H2O - the optimized HTTP/1, HTTP/2, HTTP/3 server. https://github.com/h2o/h2o. (Accessed on 12/12/2025).

[7] 2025. State of the Web: Total Number of Requests. https://httparchive.org/reports/state-of-the-web#reqTotal. (Accessed on 12/12/2025).

[8] 2025. The Trace Event Profiling Tool (about:tracing). https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/. (Accessed on 12/12/2025).

[9] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. IETF. https://datatracker.ietf.org/doc/html/rfc7540

[10] Mike Bishop. 2022. HTTP/3. RFC 9114. doi:10.17487/RFC9114

[11] 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. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/butkiewicz

[12] developer.chrome.com. 2019. Lighthouse performance scoring. https://developer.chrome.com/docs/lighthouse/performance/performance-scoring

[13] David Dittrich and Erin Kenneally. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research.* Technical Report. U.S. Department of Homeland Security.

[14] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 605–620. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[15] Piotr Gawłowicz and Anatolij Zubow. 2019. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems* (Miami Beach, FL, USA) *(MSWIM '19)*. Association for Computing Machinery, New York, NY, USA, 113–120. doi:10.1145/3345768.3355908

[16] Google. 2026. https://www.tensorflow.org/ (Accessed on 24/04/2026).

[17] Joris Herbots, Robin Marx, Maarten Wijnants, Peter Quax, and Wim Lamotte. 2024. HTTP/3's Extensible Prioritization Scheme in the Wild. In *Proceedings of the 2024 Applied Networking Research Workshop* (Vancouver, AA, Canada) *(ANRW '24)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3673422.3674887

[18] Tobias Hoßfeld, Florian Metzger, and Dario Rossi. 2018. Speed Index: Relating the Industrial Standard for User Perceived Web Performance to web QoE. In *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*. 1–6. doi:10.1109/QoMEX.2018.8463430

[19] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. doi:10.17487/RFC9000

[20] Nikhil Kansal, Murali Ramanujam, and Ravi Netravali. 2021. Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 269–287. https://www.usenix.org/conference/nsdi21/presentation/kansal

[21] 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. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kelton

[22] Sunjae Kim and Wonjun Lee. 2023. HTTP Steady Connections for Robust Web Acceleration *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3154–3163. doi:10.1145/3543507.3583550

[23] Wei Liu, Xinlei Yang, Hao Lin, Zhenhua Li, and Feng Qian. 2022. Fusing Speed Index during Web Page Loading. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 23 (feb 2022), 23 pages. doi:10.1145/3511214

[24] Haarika Manda, Varshika Srinivasavaradhan, Laasya Koduru, Kevin Zhang, Xuanhe Zhou, Udit Paul, Elizabeth Belding, Arpit Gupta, and Tejas N. Narechania. 2024. The Efficacy of the Connect America Fund in Addressing US Internet Access Inequities. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 484–505. doi:10.1145/3651890.3672272

[25] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 197–210. doi:10.1145/3098822.3098843

[26] Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. 2019. Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In *Web Information Systems and Technologies (WEBIST '19)*. https://doi.org/10.5220/0008191701300143

[27] Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. 2020. Resource Multiplexing and Prioritization in HTTP/2 over TCP Versus HTTP/3 over QUIC. In *Web Information Systems and Technologies (WEBIST '19)*. https://doi.org/10.1007/978-3-030-61750-9_5

[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 1928–1937. https://proceedings.mlr.press/v48/mniha16.html

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. doi:10.1038/nature14236

[30] Nitinder Mohan, Andrew E. Ferguson, Hendrik Cech, Rohan Bose, Prakita Rayyan Renatin, Mahesh K. Marina, and Jörg Ott. 2024. A Multifaceted Look at Starlink Performance. In *Proceedings of the ACM on Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 2723–2734. doi:10.1145/3589334.3645328

[31] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali

[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 *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 417–429. https://www.usenix.org/conference/atc15/technical-session/presentation/netravali

[33] Kazuho Oku and Lucas Pardue. 2022. Extensible Prioritization Scheme for HTTP. RFC 9218. doi:10.17487/RFC9218

[34] Darijo Raca, Dylan Leahy, Cormac J. Sreenan, and Jason J. Quinlan. 2020. Beyond throughput, the next generation: a 5G dataset with channel and context metrics. In *Proceedings of the 11th ACM Multimedia Systems Conference* (Istanbul, Turkey) *(MMSys '20)*. Association for Computing Machinery, New York, NY, USA, 303–308. doi:10.1145/3339825.3394938

[35] George F. Riley and Thomas R. Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*. Springer, 15–34. doi:10.1007/978-3-642-12331-3_2

[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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 390–403. doi:10.1145/3098822.3098851

[37] Jan Rüth, Konrad Wolsing, Klaus Wehrle, and Oliver Hohlfeld. 2019. Perceiving QUIC: do users notice or even care?. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 144–150. doi:10.1145/3359989.3365416

[38] Constantin Sander, Leo Blöcher, Klaus Wehrle, and Jan Rüth. 2021. Sharding and HTTP/2 connection reuse revisited: why are there still redundant connections?. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 292–301. doi:10.1145/3487552.3487832

[39] Constantin Sander, Ike Kunze, Leo Blöcher, Mike Kosek, and Klaus Wehrle. 2023. ECN with QUIC: Challenges in the Wild. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (Montreal QC, Canada) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 540–553. doi:10.1145/3618257.3624821

[40] Constantin Sander, Ike Kunze, Hendrik Buschbaum, and Klaus Wehrle. 2026. xPrio Source Code. https://www.github.com/COMSYS/xPrio.

[41] Constantin Sander, Ike Kunze, Dario Veltri, and Klaus Wehrle. 2025. VGPrio: Visually Guided HTTP/3 Prioritization. In *Proceedings of the 2025 IFIP Networking Conference*. IFIP.

[42] Constantin Sander, Ike Kunze, and Klaus Wehrle. 2022. Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance. In *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA '22)*. IFIP.

[43] Constantin Sander, Ike Kunze, Klaus Wehrle, and Jan Rüth. 2021. Video Conferencing and Flow-Rate Fairness: A First Look at Zoom and the Impact of Flow-Queuing AQM. In *Proceedings of the Passive and Active Measurement Conference*. Springer. doi:10.1007/978-3-030-72582-2_1

[44] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[45] sitespeed.io. 2025. Browsertime: Measure and Optimize Web Performance. https://github.com/sitespeedio/browsertime. (Accessed on 12/12/2025).

[46] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[47] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao

[48] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. 2018. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the World Wide Web Conference (WWW '18)*. https://doi.org/10.1145/3178876.3186181

[49] KaKei Wong and Lin Cui. 2023. Fine-grained HTTP/3 prioritization via reinforcement learning. *Computer Networks* 233 (2023), 109880. doi:10.1016/j.comnet.2023.109880

[50] 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* (Los Angeles, CA, USA) *(Internet QoE '17)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3098603.3098604

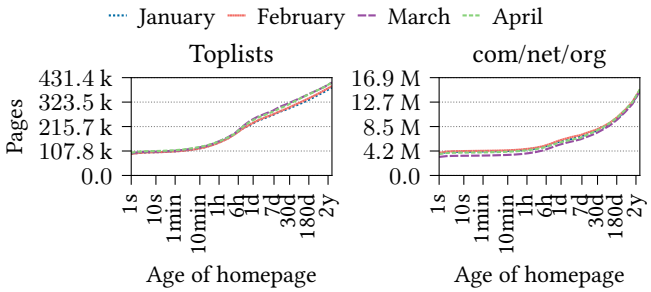


Fig. 6. Last-Modified HTTP header age. Low ages hint at dynamically generated content for which it is uncertain whether changes occurred such that costly recalculations for recent a-priori information are needed.

A Appendix

A.1 Homepage Age Measurements

For determining how often dependency information of webpages would need to be recomputed, we use the age of HTML documents as proxy. The age of a page is reset whenever a change occurs, which can indicate that the underlying dependencies changed. However, not every content change necessarily implies a dependency change. As there is no other way to check whether dependencies changed, we derive the necessity for reloads and recomputation from the page age.

For this analysis, we use large-scale measurements from a previous publication [39] to calculate the document age in Figure 6. Specifically, we conducted monthly HTTP measurements at the beginning of 2023 on all domains contained in the com/net/org domain space and the Alexa, the Umbrella, the Majestic and the Tranco Toplist. We refer to our original publication's ethical considerations regarding rate limiting and opt out.

We filter for pages that send the Last-Modified header which is used as a cache hint for whether a resource changed. We then use the header to calculate the time between request and last-modified date to get the age of the page. In total, around 20% of toplist domains and 10% of com/net/org domains set the header. Around 25% of webpages for both domain sources seem to dynamically generate their homepage with ages below a second. Moreover, toplist homepages seem to update their root document more often than com/net/org websites, with the median age being slightly more than 6 hours. As such, 50% of our checked toplist domains would require a recomputation of their homepage dependencies at least every 6h, 25% seemingly with every request.

A.2 Ethics

This paper does not raise ethical concerns as it relies on self-hosted measurements using a dataset of publicly available webpages. It does not include human subjects, user data or private information.

The age measurements in Appendix A.1 rely on measurement data of a previous publication [39]. The data does not include user information and was prepared following established ethical guidelines for internet measurements [13, 14, 39]. We refer the reader to the original publication's ethical considerations [39] that include opt out measures and rate limiting.

A.3 Further Results

A.3.1 Realworld Application of xPrio. Beside our controlled testbed results, we also investigate xPrio's performance subject to real changing network conditions. Specifically, we replace our virtual links with the Internet and a Starlink connection, where we host the webpages at our institution in Aachen, Germany and contact them via Starlink (also from Aachen, Germany).

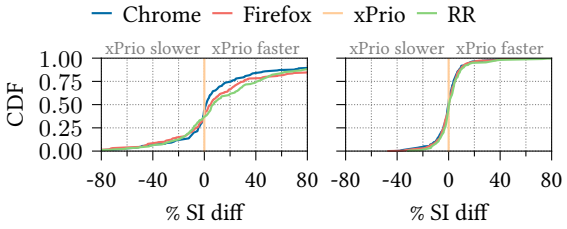


Fig. 7. xPrio Performance on Starlink Standby (left) and Starlink Residential Lite (right)

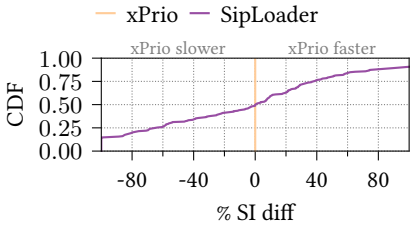


Fig. 8. xPrio Performance in Comparison to SipLoader (10 Mbps, 10 ms, 0 %)

Fig. 7 shows xPrio's SI performance when tested via Starlink in Standby and Residential Lite Mode. We can see that xPrio is able to achieve strong benefits in Standby Mode, where 30 % of websites see improvements of at least 13 % in comparison to Chrome and more than 23 %/30 % in comparison to Firefox and RR. Such strong detriments are encountered for less than 15 % of websites. As such, xPrio achieves median improvements of 2 % (Chrome) to 7 % (RR) and mean improvements of 26 % for RR, 26 % for Chrome and 29 % for Firefox.

In Residential Lite Mode, the larger bandwidth again decreases the differences. We see median improvements of 0 % and mean improvements from 0 % to 5 %.

Takeaway. *xPrio also improves performance in a low-bandwidth real-world application scenario.*

Having seen the results of xPrio w.r.t. SI and other metrics in comparison to other a-priori free strategies, we also aimed at investigating xPrio's performance in comparison to scheduling that uses a-priori information.

A.3.2 xPrio in Comparison to Related Work with A-Priori Data. In the following, we compare xPrio to an a-priori data-fueled approach, namely SipLoader [23]. SipLoader uses a client-side request scheduler fueled with a-priori visual dependency information to request visually impactful resources first. We specifically choose SipLoader [23] as it does not require Server Push, which is incompatible with our HTTP/3 browser setup (cf. Sec. 7), also optimizes SI performance, does not require extensive precomputation and code was available.

As shown in Fig. 8, SipLoader achieves benefits above 40 % for more than 30 % of webpages at 10 Mbps, 10 ms RTT and 0 % loss. However, we also see that xPrio achieves improvements above 40 % for 25 % of pages. We find that this strong divergence is rooted in SipLoader changing the layout of about 75 % of webpages, which then partially show a better performance, while for other pages, SipLoader significantly reduced performance by not accelerating critical resources. xPrio and SipLoader thus show a similar median performance, while some strong detriments of SipLoader cause xPrio to even achieve average speedups of 7 %.

Takeaway. *xPrio is not able to achieve the strong benefits of the SipLoader a-priori approach, but also avoids the strong detriments that it causes.*

A.4 Artifacts

The source code of our test framework, our simulation, our training and our server implementation is available on GitHub [40].

Received December 2025; accepted April 2026