VGPrio: Visually Guided HTTP/3 Prioritization

Constantin Sander, Ike Kunze, Dario Veltri, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University, Aachen, Germany
{sander, kunze, veltri, wehrle}@comsys.rwth-aachen.de

Abstract—**HTTP prioritization allows to signal the priority of web resources to aid and speed up the webpage loading process. However, setting optimal resource priorities is challenging. Typically, generalized priority strategies are used to achieve good performance for most websites, but the strategies struggle in certain scenarios reducing human-perceivable performance.**

Thus, we propose VGPrio, an approach that automatically optimizes resource priorities w.r.t. visual metrics / human-perceivable performance. VGPrio uses a Bayesian optimization–based method to learn prioritization strategies for websites that specifically improve the human-perceivable SpeedIndex. Through its sample-efficient method, VGPrio only requires few iterations while our evaluation on a public website corpus shows that it can improve the SpeedIndex by up to 50% compared to default strategies evading strong detriments and being more widely applicable than related work aiming at similar goals. As such, VGPrio represents a promising option to improve human-perceivable web performance beyond manual optimization.

Index Terms—**HTTP Resource Prioritization, Web Performance, SpeedIndex, Bayesian Optimization.**

I. Introduction

Modern websites consist of many resources and each has a different impact on the page-loading process and the user's visual perception [1], [2]. In particular, essential resources need to be transferred as early as possible to quickly advance the rendering state of a web page. HTTP addresses this challenge with resource priorities to send certain resources earlier than others, thus speeding up the page load [2]–[4]. Typically, the priorities are set by the browsers, which use different, generalized prioritization strategies to decide on the actual schedule and assign priorities based on a resource's type. However, each strategy has specific advantages and drawbacks for distinct websites and/or network scenarios [2], [5], [6]. While experimental approaches aim to address these weaknesses with more fine-grained generalized strategies, they can still introduce significant detriments and are not directly deployable in practice [2], [5], [7], such that a clear one-size-fits-all solution does not exist.

Accounting for this observation, research has studied the efficacy of overriding prioritization schedules on the server and using website tailored strategies instead. For instance, Reinforcement learning (RL) can be used to provide these tailored schedules [8] but typically requires vast amounts of data for training [9]. Thus, simplified simulations are used to efficiently determine PLT-related objectives that RL can improve upon. However, page load time (PLT) metrics do not correlate well with human perception, questioning the benefit of these solutions for user satisfaction.

ISBN 978-3-903176-72-0 © 2025 IFIP

In contrast, visual metrics, such as the SpeedIndex, show better correlation [10]–[12] but are harder to simulate as they require intricate knowledge of the (visual) dependencies of web page resources and their interplay with the browser rendering pipeline [13]. As a result, related work usually gathers the SpeedIndex via capturing the viewport of real browsers [2], [10], [14]. However, relying on real browsers incurs substantial computational overhead, so applying previous learning-based prioritization approaches to the SpeedIndex is infeasible. Overall, effective HTTP prioritization schemes based on visually meaningful metrics are still missing.

In this work, we fill this gap with VGPrio, a visually guided Bayesian optimization–based method for improving HTTP/3 resource priorities. VGPrio collects SpeedIndex samples taken from page loads in a real browser and uses Bayesian optimization [9] and resource clustering on these samples to derive website-specific resource schedules. Since Bayesian optimization is known to be sample efficient, VGPrio requires only a few samples for training, thus providing a feasible runtime while still relying on an expensive yet visually meaningful metric that resonates with human perception. We evaluate the performance improvements of VGPrio on a common corpus of webpages and find that VGPrio is able to improve performance for certain websites by more than 50%. In the median, it achieves around 9% improvement while significantly reducing detriments in comparison to other prioritization strategies. Hence, overall, VGPrio presents an efficient and visually guided approach to resource prioritization that can automatically adapt to web pages and can be applied to any website.

Specifically, we make the following contributions:

- We describe challenges for prioritization and why user-perceivable performance metrics are essential for solutions.
- We present how VGPrio tackles these challenges and learns prioritization strategies that improve on visual metrics.
- We evaluate VGPrio's performance on different websites and network configurations to characterize its performance in different scenarios. We show that VGPrio is able to improve performance in many scenarios.
- We discuss how to efficiently deploy VGPrio, and provide guidance concerning its extensibility and future proofness.

Structure. In Sec. II, we introduce essential web page loading mechanics and performance metrics. We then present the design of VGPrio in Sec. III and our training and evaluation methodology in Sec. IV. We evaluate the performance of VGPrio in Sec. V. In Sec. VI, we position VGPrio in the context of related work. Sec. VII discusses larger implications of VGPrio. We conclude the paper in Sec. VIII.

II. Background

Loading a web page relies on a fine-grained interplay between resource dependencies and the browser rendering pipeline. VGPrio leverages specific observations of this interaction to derive optimized resource schedules. Hence, to better position the design of VGPrio in this domain, this section describes how browsers load a web page, how resource prioritization can influence and speed up this process, and how corresponding web performance impacts can be measured.

A. Web Page Loading Process and Critical Path

Loading a web page starts with the browser downloading and parsing the main HTML document to discover the initial structure of the page and its entailed resources [15]. The browser then concurrently downloads, evaluates, and renders the new resources to eventually display the complete web page. However, certain resources, such as Javascripts or stylesheets, can change the page structure and its visible layout. Thus, to avoid inconsistent states, such resources are *blocking*, i.e., the parsing and rendering of other resources is halted until the blocking resource is downloaded and evaluated. Consequently, blocking resources and their dependencies have a crucial impact on web performance and form the *critical path*: this path needs to be processed in line before the browser can start rendering the web page and its other, non-blocking resources, such as images. Accounting for this diversity in importance for the page load, HTTP includes a dedicated mechanism for specifying which resources to transmit with which priority so that important resources can ideally be transmitted more quickly than other resources.

B. HTTP/3 Prioritization

Starting with HTTP/2 [4], browsers can signal resource priorities to web servers to inform about resource importance. While HTTP/2, originally, used a complex dependency graph [4], HTTP/3 and the most recent HTTP/2 RFC [16] remove this graph, but let browsers use the Extensible Prioritization Scheme (EPS) [3]. Specifically, every HTTP request can be assigned a priority with one out of eight urgency values and an incremental flag. Servers can then process the requests in order of ascending urgencies, where same-urgency requests are either processed round robin when the incremental flag is set or in order of arrival. To set the priority signals during loading, browsers use different heuristics [2], [6]. Chrome, e.g., follows a sequential approach by not setting the incremental flag and giving blocking resources the highest priorities. Similarly, Firefox switched to a sequential approach with HTTP/3 while it used a complex parallel approach with HTTP/2 to also load resources incrementally [6]. Other browsers used (weighted) round robin with HTTP/2 that can now only be expressed as pure round robin in the EPS. Importantly, no single heuristic (graph nor EPS-based) yields the best performance in all scenarios as a website's performance can also rely on certain non-blocking resources that the heuristics do not account for [2], [6]. For example, while prioritizing blocking resources helps progress the critical path, images may crucially shape the appearance of a page and need to be loaded quickly, especially when considering user-perceivable performance. Such nuances are covered with different performance metrics.

C. Web Performance Metrics

There are multiple metrics for gauging how fast browsers display a web page [10]–[12]. The PLT measures the time from the initial request until all resources have been fully evaluated and loaded. However, the PLT has been found to correlate poorly with human perception [10]–[12] as it ignores the visual appearance of a page. For example, above-the-fold (ATF) resources, i.e., ones that are visible in the viewport, are important for human perception, while below-the-fold (BTF) resources do not have an immediate impact; PLT gives equal weight to both kinds of resources.

In contrast, visually guided metrics consider the visual appearance of a website as a whole and, thus, implicitly the different impacts of ATF and BTF resources. For instance, the SpeedIndex [12] integrates the visual completeness of a web page over time into a single value to describe how fast the browser displays the actual web page. This approach is computationally more expensive, as it uses screenshots of a real browser loading a page to compute the visual completeness, but correlates well with human perception [10]–[12]. However, until now, the SpeedIndex and resource prioritization were considered independently and were not jointly optimized which explains the detriments that are sometimes observed [2], [6].

Takeaway. *How fast a web page loads is defined by its used resources that impact performance differently. As a result, browsers use heuristics to set HTTP priorities to load important resources quicker. Yet, these heuristics are not guided by human-perceivable performance metrics such as the SpeedIndex and can thus show detriments.*

Since web content is arguably intended for humans, VGPrio makes central use of the SpeedIndex to gain human-perceivable performance improvements, as we detail next.

III. Design

Focusing on the effects of web performance on human perception, VGPrio derives tailored per-website HTTP/3 prioritization strategies that minimize a web page load's SpeedIndex. For this, VGPrio navigates the large prioritization space of all possible resource prioritization permutations and evaluates its performance. To do so efficiently, VGPrio subdivides the space and explores it using Bayesian optimization (BO) to enable discovering strategies with the highest impact on user-perceived performance. In the following, we first give an overview of VGPrio's general design before we provide detailed information on how it optimizes the HTTP/3 priorities and how prior web page information is used to support the optimization approach.

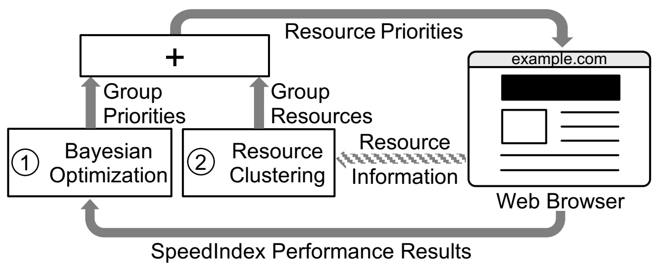


Fig. 1: VGPrio has two main components: web page-informed resource clustering is used to group resources limiting the dimensionality of our optimization space and Bayesian optimization is used to efficiently optimize the resource group priorities guided by SpeedIndex samples.

A. Overview

VGPrio is designed to optimize the SpeedIndex of a website for which it repeatedly, yet efficiently, finds and evaluates different possible prioritization schedules. Doing so, it identifies priority impacts and guides the overall strategy into the direction of the best SpeedIndex performance (cf. Figure 1). Formalized, it focuses on the optimization problem $\operatorname{argmin}_{\mathbf{x}} f(\mathbf{x})$ where $f(\mathbf{x})$ describes the SpeedIndex of a web page load with resources priorities $\mathbf{x}$. That means, VGPrio adjusts $\mathbf{x}$ to reach a configuration with an improved SpeedIndex. However, this optimization problem comes with two main challenges:

First, calculating the SpeedIndex, as described in Section II, consists of a page load including the intertwined and concurrent web page loading pipeline. Therefore, deriving a closed-form solution of $f(\mathbf{x})$ is prohibitively difficult, and the optimization problem cannot be solved analytically. Instead, $f(\mathbf{x})$ can only be seen as a *black-box function* that is also *expensive* to evaluate, as a real browser needs to fully load the web page, which can take up to several tens of seconds.

Second, the parameter space of resource priorities $\mathbf{x}$ can quickly explode. In particular, websites can easily consist of 50 or more resources (cf. Fig. 4) and each resource has eight possible urgency values plus the incremental flag (cf. Sec. II). This results in 16^n possible prioritization combinations where larger websites strain the approach and its scalability significantly, especially in light of the expensive objective. Hence, the size of $\mathbf{x}$ needs to be *limited* by grouping and excluding certain combinations to *bound complexity*.

Given these conditions, VGPrio uses two building blocks:
① **Bayesian Optimization.** VGPrio uses Bayesian optimization (BO) [9], [17] to minimize $f(\mathbf{x})$. BO specifically targets black-box functions and can learn the SpeedIndex behavior of the loading process by repeatedly sampling $f(\mathbf{x})$. Additionally, BO is sample-efficient, i.e., it requires few samples for training, which is particularly beneficial in light of the expensive page load process. In contrast, alternatives, such as deep reinforcement learning (DRL), also apply in general but require prohibitively large amounts of data.
② **Resource Clustering.** VGPrio employs a web-page-informed resource clustering to contain the parameter space of $\mathbf{x}$. While our optimization problem can generally grow nearly unlimitedly with the number of resources and BO struggles with large input spaces, many resource priority combinations are prohibitive or do not need full flexibility. For instance, deprioritizing blocking resources is detrimental and resource groups can have similar influence on the objective such that their priority combinations can be aggregated. Clustering these combinations with web page information thus allows us to reduce the space, include specific domain knowledge, and relax the problem.

In the following, we describe the building blocks in detail and show how they are connected as visualized in Figure 1.

B. Bayesian Optimization

BO optimizes black-box functions by cleverly sampling their input space to find estimated minima. For this, BO (typically) remodels the functions via Gaussian processes and uses uncertainty and value estimates to find new sampling points via a so-called acquisition function. These newly found sampling points are then used to update BO's model to improve its certainty and value estimates. This target-driven acquisition of new sampling points allows BO to be very efficient, especially when used with expensive black-box functions.

To apply BO to our use case, we model the web page loading process as a function $f(\mathbf{x})$, which yields the SpeedIndex when using prioritization $\mathbf{x}$. For now, $\mathbf{x}$ represents a vector of resource priorities, where every resource establishes two entries in the vector consisting of its urgency and the incremental flag. For sampling f, we fully load the according web page given resource priorities $\mathbf{x}$ and use the perceived web performance as result of f. Since web performance measurements can be noisy, we sample f ten times and use the average of the ten runs as the sample for BO. Applying BO with the UCB [9] acquisition function, we repeatedly evaluate the web performance of different prioritization schedules and automatically reach a prioritization $\mathbf{x}$ that minimizes the SpeedIndex.

However, when directly representing each resource in $\mathbf{x}$, $\mathbf{x}$ grows with the number of resources, such that our approach would be sensitive to a web page's size. Furthermore, BO scales badly to large input spaces. To address this challenge, we limit the input space by clustering resources, so $\mathbf{x}$ represents the priorities for groups of resources.

C. Resource Clustering

To limit the input space and avoid the curse of dimensionality, we cluster resources by their web page meaning and also remove certain groups from the input space to relieve optimization further. Specifically, we extract type and visual information of resources, which is then used as follows:

First, we combine non-blocking script resources, such as asynchronous Javascripts, into one group in the optimization vector, such that these resources receive their own priority signal. Blocking resources, on the other hand, are excluded from the optimization, as they reside on the critical path of the page, so require highest priority and get urgency 0.

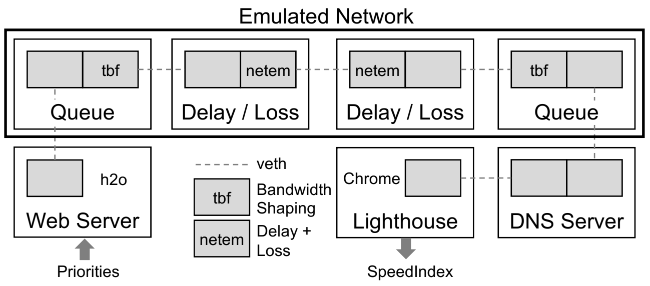


Fig. 2: Emulated network testbed for web page loading: Web server and browser reside in different namespaces that are connected via virtual links that are shaped to represent various bandwidth, loss and latency restrictions.

Second, we divide images into subgroups through k-means clustering on their height, width and whether they are ATF resources. We choose $k = 3$ for the best trade-off between cluster cohesion and input space complexity. In most cases, these three groups resemble small, big and BTF images.

Third, we combine all other resources into one group of the optimization vector for which we apply the standard priority but decide on the incremental flag.

These group priority parameters are then used as $\mathbf{x}$, forming a 9-dimensional (non-blocking urgency+incremental, 3 $\times$ images u+i, other i) input vector on which BO can optimize. For retrieving the actual resource priorities, e.g., when evaluating a newly acquired sampling point, we invert the clustering and apply the group priorities to the group resources, which then enable correctly prioritizing the HTTP resource requests.

IV. Training & Evaluation Methodology

We train and evaluate VGPrio in a reproducible testbed environment similar to MahiMahi [18]. In particular, we first download websites, then host them on a local web server, and, finally, access them with a browser through a virtualized network with configurable conditions. The resulting SpeedIndex in this process is then used for our final evaluation but also for training. In the following, we describe this process and the different components of our training and evaluation pipeline in more detail in Sec. IV-A. In Sec. IV-B, we then describe our evaluation scenarios.

A. Training & Evaluation Testbed

We use a training and evaluation methodology that allows reproducibility of our results. In particular, we use a focused testbed with well-controlled network conditions that locally replays target websites and captures web performance metrics through Lighthouse [19].

Overview. At its core, our testbed consists of an adapted h2o web server, an emulated network, a DNS server, and a local browser embedded in the Lighthouse web performance measurement framework as shown in Fig. 2.

Web Page Replay. To conduct experiments on a specific website, we first download the website and store all entailed resources as well as the request/response headers used for the transfers. We then host the website on our h2o web server, which can enforce different prioritization strategies. We request the websites using Chrome 95.0.4638.54 from within Lighthouse to extract meaningful web performance metrics, such as the SpeedIndex, and configure Lighthouse to represent a local desktop browser while also disabling its internal network simulation, which is known to provide unsteady estimates [20]. Instead, we explicitly control the network conditions with our emulated network.

Network Emulation. For connecting browser and web server, we deploy an emulated network using virtual ethernet links and network namespaces. This setup allows us to configure different network conditions including bandwidth, round-trip time (RTT), and packet loss via Linux `tc`, `netem`, and `tbf`. Specifically, we dedicate different namespaces to different functions to avoid backpressure / influences between the `tc qdiscs` and the network stack biasing its function [21], [22]. The testbed is deployed on a consumer-grade PC equipped with an Intel i5-4590 CPU and 16 GB of RAM, representing consumer-perceived performance.

Training. For training, we use our testbed to gather SpeedIndex samples of a web page for differing prioritization schedules which VGPrio can then use to optimize on. However, before starting the actual training, we first load the web page once and execute a custom Javascript to gather web page resource information by extracting images and script tags from the page as well as specific features, such as image positions. We then feed this information into VGPrio's resource clustering to generate the resource groups on which BO operates. Subsequently, we also run Lighthouse 30 times with the default prioritization to estimate the variance of the measurements, priming the BO model on the expected noise.

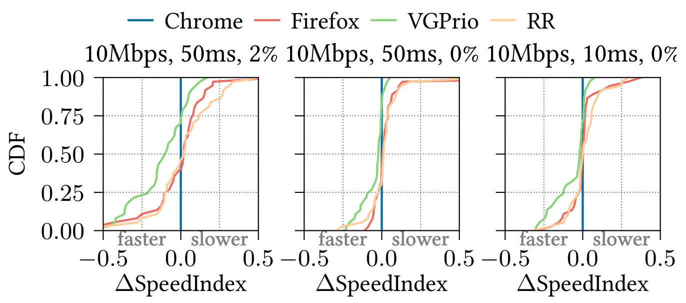
After this setup, we commence VGPrio's actual optimization process. Starting with a first random schedule, we let VGPrio generate a new priority schedule via BO which we configure on our web server. We then measure the SpeedIndex for the given web page ten times and take the mean of these ten measurements (to even out noise) as sample for BO. For each website, we perform 50 iterations of this process, aiming to minimize the SpeedIndex and improve the perceivable performance resulting in a final prioritization strategy.

Evaluation. For evaluation, we again use our testbed to gather SpeedIndex samples of a web page subject to the optimized strategy and compare it against prioritization schedules defined by the other prioritization strategies. Specifically, we compare the strategies in the scenarios that we define next.

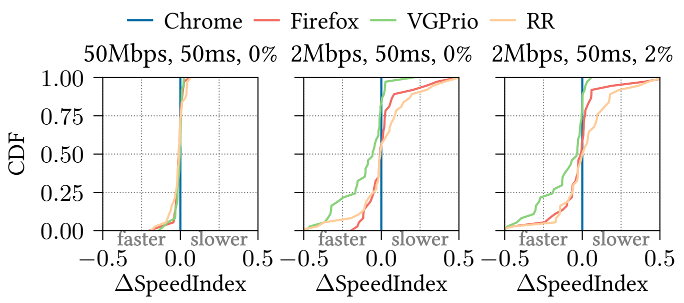
B. Training & Evaluation Scenarios

We train and evaluate VGPrio on different websites and in different network scenarios and compare its performance to different prioritization strategies.

Website Corpus. For our evaluation, we rely on a website corpus originally proposed by Wijnants et al. [2]. Specifically, the corpus consists of 40 popular websites with different sizes, numbers of resources, and complexities, thus equally representing simple but also resource-intensive web pages. We



(a) Performance for 10Mbps with varied RTTs and Loss



(b) Performance for varied bandwidth

Fig. 3: Relative SpeedIndex changes of VGPrio, round robin and Firefox's strategy against Chrome's strategy.

discard one internal web page and www.bitly.com from the corpus due to unavailability and replay issues. Otherwise, we use an exact copy of the web pages with unaltered HTTP headers and body contents, including possible compression used by the sites.

Network Configurations. We evaluate VGPrio in a broad range of network scenarios with different bandwidths, RTTs, and loss patterns. Specifically, we use bandwidths from $2\,\mathrm{Mbps}$ to $50\,\mathrm{Mbps}$, RTTs from $10\,\mathrm{ms}$ to $50\,\mathrm{ms}$, and consider packet loss of up to $2\,\%$.

Prioritization Strategies. We compare VGPrio's performance to three prioritization strategies: round-robin scheduling, Chrome's prioritization strategy, and an HTTP/3 compatible adaption of Firefox's parallel strategy used for HTTP/2 [6]. We specifically use this adaption, as Firefox introduced a unique parallel strategy with HTTP/2 (cf. Sec. II) that it discarded for a sequential strategy very similar to Chrome with HTTP/3 [6]. As such, we use the parallel strategy to still represent a wide variety of prioritization schedules. To gain the specific priorities, we load every web page once via Chrome and also via Firefox to gather the sent prioritization signals and then use the curated signals to override the prioritization with our adapted h2o server.

Experiments. We repeat all evaluation runs 30 times and we show the corresponding median in our plots. Our artifacts are available at `https://github.com/COMSYS/VGPrio`.

V. Evaluation Results

In this section, we present the page load performance results of VGPrio across multiple representative scenarios. First, in Sec. V-A, we compare the performance of VGPrio to the default strategy of Chrome, the adapted Firefox strategy, and a plain round-robin strategy. Thereafter, we delve deeper into the behavior of VGPrio and identify specific website characteristics which VGPrio can benefit the most in Sec. V-B. We further characterize the learning progress of VGPrio in Sec. V-C and, finally, compare VGPrio to SipLoader [23], a related approach, on a subset of our scenarios in Sec. V-D.

A. *Page Load Performance*

For our initial assessment, we use the performance of Chrome's default prioritization as our baseline and compute the relative SpeedIndex improvement of VGPrio's, Firefox's and the round-robin strategy. Fig. 3 shows corresponding CDFs for experiments in six representative network scenarios. We distinguish between three base scenarios that use a bandwidth of $10\,\mathrm{Mbps}$ with different RTTs and loss rates in Fig. 3a and three diversified scenarios for performance and prioritization in Fig. 3b, where we also vary the bandwidth. In each plot, values below 0 on the left side represent relative performance improvements compared to Chrome, whereas values above 0 on the right represent deteriorations.

Base Scenarios ($10\,\mathrm{Mbps}$)**.** Starting with the best base scenarios in Fig. 3a, we find that VGPrio improves the median SpeedIndex compared to Chrome by $10\,\%$ (mean $13\,\%$) in the left-most setting, i.e., for a bandwidth of $10\,\mathrm{Mbps}$, an RTT of $50\,\mathrm{ms}$, and $2\,\%$ packet loss. In contrast, Firefox and round robin decrease performance by $2\,\%$ in the median where round robin shows weaker improvements but stronger detriments than Firefox in total. As such, VGPrio outperforms Chrome as well as Firefox and round robin. Moreover, we identify performance improvements of more than $15\,\%$ for $30\,\%$ of the websites and that performance never degrades by more than $17\,\%$. In particular, this worst-case degradation only occurs for a single website, while $90\,\%$ of websites do not experience a detriment above $6\,\%$. We attribute the strong detriment to the probabilistic packet loss increasing noise such that VGPrio is more challenged finding the model, which could be improved with more samples for averaging at cost of longer training.

For the same scenario but without packet loss (center), we see slightly weaker improvements, but also weaker detriments as transmissions are generally smoother. VGPrio still outperforms Chrome, Firefox, and round robin. More than $20\,\%$ of websites see improvements above $10\,\%$ and the maximum detriment is capped at $5\,\%$. Similarly, the performance benefits persist with a lower RTT of $10\,\mathrm{ms}$ (right), showing that VGPrio can effectively improve performance at $10\,\mathrm{Mbps}$. Next, we investigate if these benefits also translate to other bandwidths and more challenging scenarios.

Diverse Scenarios. Fig. 3b (left) shows VGPrio's performance for a bandwidth of $50\,\mathrm{Mbps}$ and an RTT of $50\,\mathrm{ms}$ with no packet loss. In this scenario, all prioritization schedules show very similar performance. We attribute this effect to the higher bandwidth moving the bottleneck to the processing side of web

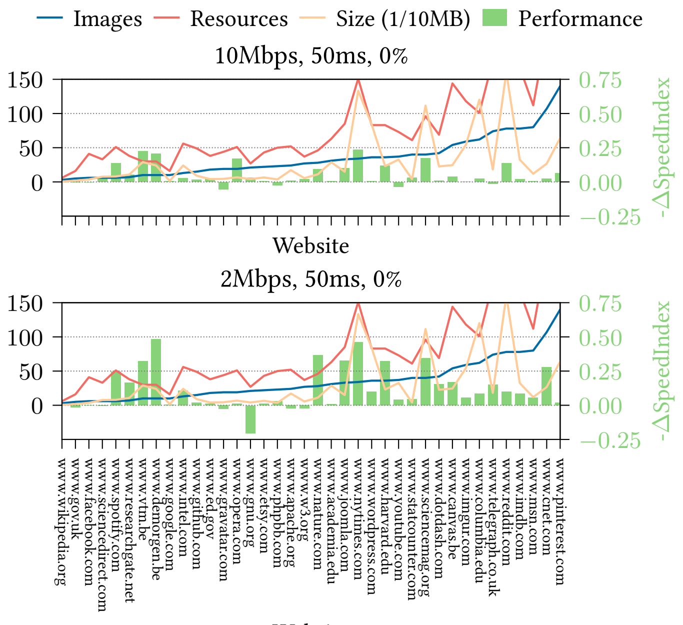


Fig. 4: VGPrio improvements per website compared to number of images and resources (sorted by number of images).

page loading, such that network scheduling does not show strong improvements, but also no detriments.

For lower bandwidths at 2 Mbps (Fig. 3b, center and right), we see stronger impacts. The network presents a strong bottleneck which VGPrio can alleviate by more than 50 % for some websites while avoiding the strong detriments that Firefox and round robin incur (more than 40 % of the websites have up to 50 % deterioration). For 2 Mbps, 50 ms RTT and 0 % packet loss, VGPrio improves the median SpeedIndex by ∼6 % (mean 11 %) and shows benefits for more than 80 % of the websites. These benefits grow beyond 25 % improvement for 20 % of the websites.

When adding 2 % packet loss, VGPrio shows a similar performance as before. The median improvement reduces to 3.5 % (mean 12 %), but again more than 75 % of the websites see benefits and 20 % achieve improvements above 27 %.

Takeaway. *All in all, our results show that VGPrio improves median SpeedIndex performance w.r.t. Chrome and avoids strong detriments that Firefox's or round-robin scheduling introduce. For lower bandwidths or higher packet loss, VGPrio improves more than 75 % of websites by up to 50 %.*

Having studied the general performance of VGPrio and in which network scenarios websites can benefit the most, we next focus on the impact of specific website characteristics on performance, aiming to provide further guidance regarding ideal deployment scenarios for VGPrio.

B. Web Page Performance Impacts

The websites in our website corpus are diverse and range from simple to resource-intensive ones. We, thus, next analyze which websites can benefit the most. For this, Fig. 4 shows the inverted SpeedIndex difference of VGPrio compared to Chrome (-ΔSpeedIndex) for each website in relation to the number of images and resources and the overall size. In contrast to Sec. V-A, values above 0 represent performance improvements, whereas values below 0 represent deteriorations.

Our first main finding is that the achievable performance benefits generally increase when more images reside on a page in the 2 Mbps case. In particular, pages with many images, such as imdb.com, but also many resources, such as nytimes.com, benefit from the targeted prioritization of resources for acceleration. We conjecture that VGPrio correctly distributes the available bandwidth to the important resources, while delaying unimportant ones.

For a higher bandwidth of 10 Mbps, the resource count impact vanishes and pages with few images benefit similarly to pages with more images. Still, especially pages with a large overall size benefit. Here, we conjecture that VGPrio correctly determines the specific impact of the (few) resources on performance and delays unimportant resources that sensitively block bandwidth due to their relatively high size.

Takeaway. *We derive that VGPrio benefits pages with many resources and images most, where correct prioritization pays out, but also pages with relatively big resources where single wrong prioritization decisions can sensitively delay page load.*

C. Training Progress

Focusing on the applicability of VGPrio, we next study its training effort. In particular, we evaluate how the SpeedIndex improvements progress with the number of training iterations to assess how many iterations might be needed in practice to generate a suitable prioritization schedule.

Fig. 5 shows the SpeedIndex progress of VGPrio for every run across all prior scenarios where we plot VGPrio's training SpeedIndex values scaled to the range from 0 to 1 based on the minimum/maximum observed, i.e., lower values represent a better performance. The red line describes the mean of all training results per iteration.

As can be seen, VGPrio's training shows an asymptotic behavior where 15 iterations already suffice to reach 25 % of the overall reached minimum. Hence, our results indicate that VGPrio does not need the full 50 iterations we used during training and instead allows a trade-off between iterations and performance which also mirrors in its runtime.

Runtime Impact. Since VGPrio relies on real web browser measurements, every iteration represents a real web page

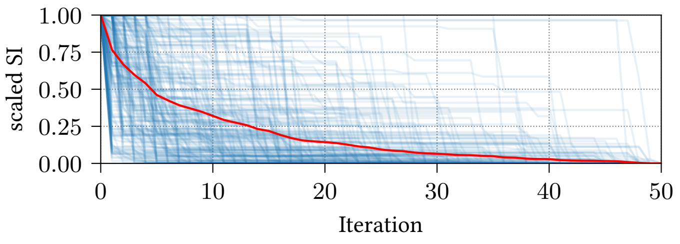


Fig. 5: Training progress / performance of VGPrio per iteration for all scenarios scaled by minimum/maximum.

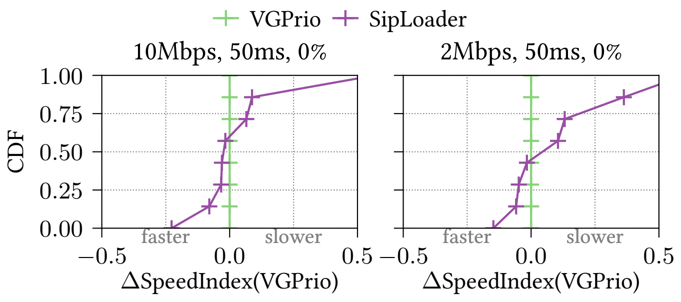


Fig. 6: Relative SpeedIndex changes of related work SipLoader in comparison to VGPrio.

load via Lighthouse with additional processing steps, e.g., for extracting the SpeedIndex. We found a median Lighthouse runtime of 32 s that translates to a total of approximately 4.5 h of measuring the SpeedIndex with 50 iterations and ten repeated measurements per iteration. Additionally, our network emulation restarts with every repetition for accuracy to reset any potential state. However, this restart can be omitted leaving Lighthouse as main contributor. Yet, as 15 iterations suffice to reach 75 % of VGPrio's performance, this runtime could be reduced to below 1.5 h or 10 min when parallelizing the individual measurements. In comparison, the actual runtime of BO and our clustering is negligible.

Takeaway. *While the reliance on real browser measurements introduces costly computations, VGPrio's use of BO and clustering for reduced parameter spaces allows it to still achieve good results with few iterations, allowing effective training times below* 10 min *when fully parallelizing the measurements.*

D. Comparison with Related Work

Besides comparing VGPrio to standard industry prioritization strategies, we also compare VGPrio with the publicly available and also visually guided web accelerator SipLoader [23]. In short, SipLoader extracts visual dependencies of websites and integrates a Javascript resource scheduler into web pages that starts resource requests on the client side, preferring visually impactful resources. It specifically does not use a learning-based method that optimizes the overall SpeedIndex, but a greedy approach, which loads most impactful resources first, representing a contrasting solution to VGPrio. Thus, we reran the provided analysis pipeline of SipLoader on our website corpus and reran our measurements for comparing the approaches. However, we noticed that 20 of our 38 web pages experienced a strong change in their visual appearance with missing elements. We observed similar effects in SipLoader's original artifacts. For ten more pages, we saw slight design changes caused by SipLoader rewriting web page content. Thus, we compare VGPrio and SipLoader only on the remaining eight pages that were not impacted.

In Fig. 6, we use VGPrio as reference and show SipLoader's relative SpeedIndex difference. As can be seen, SipLoader achieves improvements for some of the websites but also detriments for others. Median differences are around −2.5 % and 4 % for our 10 Mbps and 2 Mbps cases with means above 4 %/11 %. We deduce the improvements to SipLoader's specific advantage of deciding when a request starts, while VGPrio ultimately relies on the browser starting the request to then prioritize it. On the other hand, we deduce the performance disadvantages of SipLoader (aside its web page rewriting stage) to following a greedy, per-element optimization approach. We object that this approach neglects the interplay of all resources, which defines the joint visual appearance and, thus, the overall SpeedIndex. Moreover, SipLoader cannot control prioritization features such as incremental streams which can help to load progressive images in parallel. Hence, performance-wise, there is no clear winner among the two approaches. However, considering the error-prone nature of SipLoader, which changes the look of many web pages, we conclude that VGPrio presents an equally beneficial, but more robust and widely applicable solution to improve human-perceivable performance.

Takeaway. *Related work using visual impact for Javascript-based scheduling achieves similar results to VGPrio, but is less robust and, thus, not as widely applicable as VGPrio.*

VI. Related Work

With its focus on HTTP prioritization, VGPrio contributes to a longer line of research on improving web performance. We distinguish three main branches of work, focusing on Server Push, client-side request scheduling, and HTTP prioritization.

Server Push Optimization. Server Push is an HTTP feature that enables web servers to initiate unsolicited resource transfers prior to requests to improve performance. In general, related work on server push finds that simply pushing all available resources is detrimental for performance [14], [24] and that targeted push instrumentation is required. Hence, Vroom [25] implements a dependency resolution process to identify critical resources and push these independent of the other resources. Similarly, HTTP Steady Connections [26] analyzes web page dependencies to identify when a browser is waiting for a specific resource, aiming to provide resources in a way that they are fully available when needed. Alohamora [14] uses DRL combined with PLT simulations to find network- and website-specific push strategies. Klotski [27] and Webgaze [28] gather user preferences for resource priority (manually and via eye-tracking), and then specifically push the identified resources. All approaches have in common that they either require manual intervention or rely on the PLT. Hence, they do not offer automated and human-perception-guided optimization. Furthermore, Server Push has been deprecated by Chrome [29] and Firefox [30], due to its unclear benefits and drawbacks, such that more than 80 % of web users [31] cannot use it. In contrast, VGPrio automatically improves on the SpeedIndex and is independent of client support, as it overrides the signal at the server which subsequently needs to decide on requests scheduling either way. Hence, even deprecating HTTP prioritization in clients would not render VGPrio inapplicable.

Client-Side Request Scheduling. Another branch of work shifts the request scheduling to the client with a Javascript-based scheduler that overrides the actual web page, sends the required requests at the anticipated times, and reconstructs the page. For instance, Polaris [32] dissects Javascript code and identifies whether two scripts access the same resources. Thereafter, it decides, whether execution needs to be blocked and waiting for a script is required, or whether blocking can be skipped. SipLoader [23] gathers the visual impact of resources to then request visually important requests first. Both approaches alter web pages to implement their schedulers, which is known to be error-prone [33], and we indeed saw issues when applying SipLoader in our evaluation (cf. Sec. V-D). Additionally, the scheduler incurs overhead w.r.t. transfer size and computation. In contrast, VGPrio does not require additional Javascript files and does not alter web pages, but instead alters the server-side prioritization signals.

HTTP Prioritization. In the field of HTTP prioritization, different works aim for generalized strategies for more fine-grained resource scheduling [2], [5], [7], while still being universally applicable to a broad set of websites. This website-agnostic view does not reap all of the available performance benefits and it can also have detrimental effects, as we have seen for round robin and Firefox. As an alternative, DRP-RL [8] uses DRL for generating website-specific resource priorities. The approach uses an objective that is guided by the PLT and how long critical resources were blocked which is simulated to account for the increased amounts of data needed for DRL. Hence, DRP-RL does not directly focus on human-perceivable performance and relies on simulations, which also makes it hard to adapt to visual metrics. With VGPrio, we specifically avoid these constraints by relieving the need for high amounts of data via BO and clustering to directly use SpeedIndex samples from real browsers.

VII. Discussion

After evaluating the performance of VGPrio and positioning it in the context of related work, we finally discuss VGPrio with a specific focus on deployability and future use cases.

A. Deploying VGPrio

Our VGPrio proof-of-concept implementation shows that website-specific priority strategies can achieve good performance, while avoiding detriments (as, e.g., caused by the generalized strategies) or limiting applicability (e.g., due to forcefully rewriting web pages as done by SipLoader). However, VGPrio also comes with a higher complexity than the generalized strategies. Still, this complexity does not rule out a larger-scale deployment and neither do seemingly prohibitive training cost and model selection as we discuss next.

Training Cost. The training phase of VGPrio can take several hours. Moreover, in theory, every web page requires its own training, incurring a high computation penalty that is not necessary with the traditional prioritization strategies. However, subpages of a website typically follow a similar structure [14] that VGPrio captures. Thus, retraining is not needed for such website subpages as long as VGPrio can distinguish and associate the resources. This is the specific task of VGPrio's resource clustering, which can be adapted accordingly to map resource information between such pages. For example, we could load each subpage once to determine the resource mapping or we could leverage information contained in common content management systems without needing a full reload [14]. Further generalizing this consideration, VGPrio does not necessarily require retraining for all web page changes, but only when the structure is changed. Additionally, the training process can be sped up by distributing the page load process to multiple machines as demonstrated in Sec. V-C.

Model Selection. Besides training, VGPrio also needs to select the correct model for each scenario. Selecting the correct model per web page is trivial, but also requires network condition estimates, such as bandwidth, RTT, and loss. Notably, these estimates can be gathered a-priori from public user speedtest data [34], [35] (for bandwidth and loss) or at connection establishment (for RTT) to select the closest model.

Inference Cost. Generating resource priorities on the server running the full VGPrio pipeline is not excessively expensive, but potentially unwelcome when processing huge amounts of requests per second. Yet, it is not necessary to run the full pipeline every time, but it is possible to cache its results, saving computation and freeing resources. The gathered priority information can then be set by webservers or CDNs without fully executing VGPrio.

B. Future Effect of Prioritization

Our evaluation shows that VGPrio is limited in its effectiveness by resource prioritization being actually effective. With high bandwidths, we could see that the performance bottleneck shifted away from the network to client-side processing. In this scenario, prioritization and, thus, also VGPrio were less effective. Yet, we argue that web pages will further grow in resource count and size in the future, as has happened in the past [1], straining the network again. Additionally, client processing power will also grow, again focusing performance issues on the network side and how resources are ultimately transferred. As such, we argue that VGPrio will be able to also affect higher bandwidths when processing power and web page weight increase. Moreover, we think that lower bandwidth scenarios are also of interest given that median bandwidths in mobile networks are in the range of 15Mbps [36] and adversarial scenarios with high packet loss can still happen.

C. Other Metrics and Future Web Features

As VGPrio models the overall objective as a black-box function, it is independent of the actual metrics used to define the aim of optimization. Hence, besides the SpeedIndex, VGPrio can also use other metrics, e.g., the Largest Contentful Paint (LCP), as its optimization goal. The independent modeling and the sample-efficiency allow for freedom in which metrics are used with no need for simulations. As such, we argue that VGPrio is also easily adaptable to future web features: updating the browser suffices for the optimization to account for new features that would otherwise require intricate domain knowledge and reconfiguration of performance simulations.

VIII. Conclusion

In this paper, we present VGPrio, an approach to create and learn website-optimized prioritization strategies that directly improve the SpeedIndex, a visual, human-perceivable web performance metric. As the SpeedIndex is hard to simulate, VGPrio uses real web browser measurements that, however, incur a significant cost. Thus, VGPrio introduces a sample-efficient Bayesian optimization-based learning approach in combination with web resource clustering to both reduce the parameter space and efficiently scan it for feasible training. Our results indicate that VGPrio can significantly reduce detriments of traditional prioritization strategies while retaining and even improving the performance. In particular, VGPrio improves the mean performance for many websites by more than $10\,\%$ with some websites seeing performance benefits of above $50\,\%$ for low bandwidths. At the same time, VGPrio does not need to alter web page content, making it more easily applicable than related work, and its training time can be reduced to few minutes. As such, VGPrio presents a valuable new method for faster web page loads.

Acknowledgments

This work has been funded by the German Research Foundation DFG under Grant No. WE 2935/20-1 (LEGATO). We thank the anonymous reviewers for their valuable feedback and Hendrik Buschbaum for facilitating measurements with SipLoader in our testbed.

References

[1] HTTP Archive, "State of the Web: Total Number of Requests," https://httparchive.org/reports/state-of-the-web, 2024, (Accessed on 07/03/2025).

[2] M. Wijnants, R. Marx, P. Quax, and W. Lamotte, "HTTP/2 Prioritization and Its Impact on Web Performance," in *World Wide Web Conference*. ACM, 2018.

[3] K. Oku and L. Pardue, "Extensible Prioritization Scheme for HTTP," RFC 9218, Jun. 2022.

[4] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," IETF, RFC 7540, 2015.

[5] R. Marx, T. De Decker, P. Quax, and W. Lamotte, "Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC," in *Web Information Systems and Technologies (WEBIST '19)*, 2019.

[6] C. Sander, I. Kunze, and K. Wehrle, "Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance," in *Network Traffic Measurement and Analysis Conference*. IFIP, 2022.

[7] R. Marx, T. De Decker, P. Quax, and W. Lamotte, "Resource Multiplexing and Prioritization in HTTP/2 over TCP Versus HTTP/3 over QUIC," in *Web Information Systems and Technologies (WEBIST '19)*, 2020.

[8] K. Wong and L. Cui, "Fine-grained HTTP/3 prioritization via reinforcement learning," *Computer Networks*, vol. 233, p. 109880, 2023.

[9] R. Garnett, *Bayesian Optimization*. Cambridge University Press, 2023.

[10] T. Zimmermann, B. Wolters, and O. Hohlfeld, "A QoE Perspective on HTTP/2 Server Push," in *Workshop on QoE-Based Analysis and Management of Data Communication Networks*. ACM, 2017.

[11] J. Rüth, K. Wolsing, K. Wehrle, and O. Hohlfeld, "Perceiving QUIC: do users notice or even care?" in *Conference on Emerging Networking Experiments And Technologies*. ACM, 2019.

[12] T. Hoßfeld, F. Metzger, and D. Rossi, "Speed Index: Relating the Industrial Standard for User Perceived Web Performance to web QoE," in *International Conference on Quality of Multimedia Experience*, 2018.

[13] E. Bocchi, L. De Cicco, and D. Rossi, "Measuring the Quality of Experience of Web users," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, p. 8–13, Dec 2016.

[14] N. Kansal, M. Ramanujam, and R. Netravali, "Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly," in *Networked Systems Design and Implementation*. USENIX, 2021.

[15] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying Page Load Performance with WProf," in *Networked Systems Design and Implementation*. USENIX, 2013.

[16] M. Thomson and C. Benfield, "HTTP/2," RFC 9113, Jun. 2022.

[17] J. B. Mockus and L. J. Mockus, "Bayesian approach to global optimization and application to multiobjective and constrained problems," *Optimization Theory and Applications*, vol. 70, no. 1, pp. 157–172, 1991.

[18] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate Record-and-Replay for HTTP," in *Annual Technical Conference*. USENIX, 2015.

[19] "Introduction to Lighthouse – Chrome for Developers," https://developer.chrome.com/docs/lighthouse/overview, (Accessed on 05/03/2025).

[20] Matt Zeunert, "How does Lighthouse simulated throttling work?" https://calendar.perfplanet.com/2021/how-does-lighthouse-simulated-throttling-work/, 2021, (Accessed on 07/03/2025).

[21] "Best Practices for Benchmarking CoDel and FQ CoDel," https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel/#the-netem-qdisc-does-not-work-in-conjunction-with-other-qdiscs, (Accessed on 02/03/2025).

[22] "tc-netem - Limitations," https://man.archlinux.org/man/tc-netem.8.en#LIMITATIONS, (Accessed on 02/03/2025).

[23] W. Liu, X. Yang, H. Lin, Z. Li, and F. Qian, "Fusing Speed Index during Web Page Loading," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3511214

[24] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle, "Is the Web Ready for HTTP/2 Server Push?" in *Conference on Emerging Networking EXperiments and Technologies*. ACM, 2018.

[25] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha, "Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution," in *Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[26] S. Kim and W. Lee, "HTTP Steady Connections for Robust Web Acceleration," in *World Wide Web Conference*. ACM, 2023.

[27] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices," in *Networked Systems Design and Implementation*. USENIX, 2015.

[28] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das, "Improving User Perceived Page Load Times Using Gaze," in *Networked Systems Design and Implementation*. USENIX, 2017.

[29] "Removing HTTP/2 Server Push from Chrome," https://developer.chrome.com/blog/removing-push/, 2022, (Accessed on 07/03/2025).

[30] "Firefox 132 for developers," https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/132, 2024, (Accessed on 06/03/2025).

[31] Statcounter, "Browser Market Share Worldwide," https://gs.statcounter.com/browser-market-share#monthly-202404-202407-bar, 2024, (Accessed on 07/03/2025).

[32] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster Page Loads Using Fine-grained Dependency Tracking," in *Networked Systems Design and Implementation*. USENIX, 2016.

[33] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin, "Flywheel: Google's Data Compression Proxy for the Mobile Web," in *Networked Systems Design and Implementation*. USENIX, 2015.

[34] Measurement Lab, "The M-Lab Network Diagnostic Tool Data Set," https://measurementlab.net/tests/ndt, (Accessed on 07/03/2025).

[35] C. Dovrolis, K. Gummadi, A. Kuzmanovic, and S. D. Meinrath, "Measurement lab: overview and an invitation to the research community," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, p. 53–56, Jun. 2010.

[36] R. Sanchez-Arias, L. G. Jaimes, S. Taj, and M. S. Habib, "Understanding the State of Broadband Connectivity: An Analysis of Speedtests and Emerging Technologies," *IEEE Access*, vol. 11, pp. 101 580–101 603, 2023.