**Title**
Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On-the-Fly



**Author**
Kansal, Nikhil

**Publication Date**
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

# Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On-the-Fly

A thesis submitted in partial satisfaction
of the requirements for the degree Master of Science
in Computer Science

by

Nikhil Kansal

2019

ABSTRACT OF THE THESIS

Alohamora: Reviving HTTP/2 Push and Preload
by Adapting Policies On-the-Fly

by

Nikhil Kansal

Master of Science in Computer Science
University of California, Los Angeles, 2019
Professor Ravi Arun Netravali, Chair

Despite their promise of improved performance, HTTP/2's server push and link preload features have seen minimal adoption, largely because designing performant push/preload policies requires complex reasoning about the subtle relationships between page content, browser state, device resources, and network conditions. Static policies and guidelines that sufficiently generalize across these diverse conditions remain elusive.

We present Alohamora, a system that automatically generates push/preload policies using Reinforcement Learning (RL). Alohamoratrains a neural network that, given inputs that characterize the page structure and execution environment, outputs a push/preload policy for the page load at hand. To ensure efficient and practical training despite the large space of potential policies, number of pages served by a given site, and high mobile page load times, Alohamoraintroduces several key innovations: a faithful page load simulator that can evaluate a policy in several milliseconds (compared to 10s of seconds for a regular page load), and a page clustering strategy that appropriately balances insights for push/preload with the number of pages required during training. Experiments across a wide range of pages and mobile execution environments reveal that Alohamorais able to accelerate page loads by 19-57% and 12-34% for page load time and Speed Index, respectively.

The thesis of Nikhil Kansal is approved.

Harry Guoqing Xu

Todd D Millstein

Ravi Arun Netravali, Committee Chair

University of California, Los Angeles

2019

This work is dedicated to my loving mother, father, and sister, who have supported me throughout my years of education with unconditional love and guidance.

# Contents

## List of Figures

## List of Tables

Acknowledgements

This work was produced under the close advisement of Professor Ravi Netravali, who provided research direction and guidance in producing a submission of this system to MobiSys 2020. Murali Ramanujam was also a core contributor to the evaluation of the system, the implementation of capturing viewport-critical requests, and some written portions of the paper.

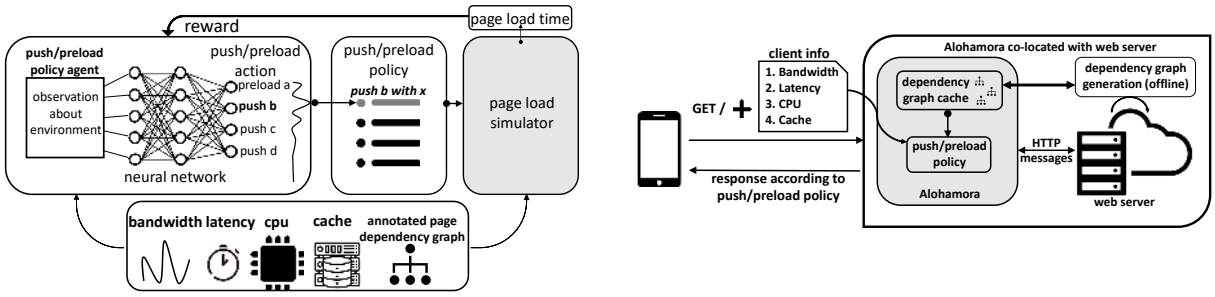
# 1 Introduction

Mobile web browsing has rapidly grown in popularity, and recently surpassed its desktop counterpart in terms of global web traffic share [3, 4, 5]. Given the importance of mobile web speeds for both user satisfaction [6, 7, 8] and content provider revenue [9], a vast array of optimizations have been developed to improve mobile page loads [10, 1, 11, 12, 13, 2, 14]. Yet performance continues to fall short of user expectations in practice. Even on a state-of-the-art mobile phone and LTE cellular network, the median page still takes over 10 seconds to load [15, 1].

Recent studies have identified that a key culprit to slow mobile page loads is the blocking network delays that arise from the dependencies between the objects on a page [1, 10]. For example, a browser may learn that it must load an image only after fetching and executing a JavaScript file, which is discovered only after downloading and parsing the page's top-level HTML. Such dependency chains essentially serialize object fetches, which in turn results in high load times, particularly on mobile networks where access link latencies tend to be high [16, 17].

The latest HTTP/2 web standard [18] anticipated the negative impact that network delays have on web performance, and in response, includes several relevant optimization features. Most notable are HTTP/2 *push* and *preload*. With push, servers can proactively send resources to clients in anticipation of future requests; requests for already-pushed resources can be satisfied locally at the client, avoiding blocking network fetches. In contrast, with preload, servers can notify clients of resources that they will soon require (potentially from other domains) by listing those URLs in HTTP headers. Clients issue requests for those resources immediately after parsing HTTP headers, and without evaluating response bodies, thereby parallelizing network fetch and object computation tasks [1].

Unfortunately, despite the promise of HTTP/2's push/preload features, developing performant push/preload policies has proven to be challenging, leading to low adoption rates. For example, we find that only 5% of the Alexa top 500 pages [19] include a domain that uses push or preload; this drops to 0.9% for the Alexa top 10,000 pages. A major reason is that the performance of




(a) Offline training.

(b) Online inference.

Figure 1: Alohamora trains its HTTP/2 push/preload policy generation model using Reinforcement Learning, exploring a large search space of environmental resources and push/preload policies, and learning from the resulting (simulated) performance. During client page loads, Alohamora collects the required inference inputs from unmodified client browsers and servers which track changes to their page dependency graphs; the generated push/preload policies are applied transparently for the remainder of the load.

a given push/preload policy depends on the subtle, low-level interactions between page content, browser (cache) state and execution dependencies, client device and network resources, and QoE goals [20, 21, 22, 23]. Consequently, even for a given page, we find that using a policy outside of the execution environment for which it was designed can either forego significant (18-31%) performance benefits or degrade performance by up to 20% compared to a default browser (§2).

These results preclude the static policies and guidelines promoted by prior push/preload systems [1, 21, 22], and instead highlight the need for *dynamic, adaptive* policies that explicitly target the environments in which they are deployed. For example, the aggressive push/preload policies that effectively utilize resources in high-bandwidth settings must be shrunk or dispersed across a page load as link rates drop to avoid potential contention that slows down the downloads of blocking resources. Similarly, as device CPU speeds decrease, policies should grow to take advantage of the (increased) blocking compute delays that leave the network idle.

In this paper, *we ask whether a machine-learning system can learn and dynamically tune push/preload policies for different pages and execution environments*. We present **Alohamora**, a web optimization system that learns push/preload policies entirely through experience using Reinforcement Learning (RL) (Figure 1). Alohamora represents its policy generation control logic as an

expressive neural network (trained offline to avoid exposing real users to suboptimal push/preload policies). At inference time (i.e., during client page loads), Alohamora's model takes as input a set of features that summarize the client's execution environment (network, CPU speed, cache contents), and structural information about the page at hand, and outputs a push/preload policy intended to optimize QoE in the current page load. Importantly, Alohamora's input data collection operates with *unmodified* client browsers, and requires minimal server-side changes: servers provide structural information about their pages (which content management systems commonly track [24, 25]), and Alohamora's policy generation runs (transparently) on a co-located frontend server.

Realizing Alohamora's data-driven approach to HTTP/2 push/preload policy generation requires overcoming two key practical challenges with respect to training efficiency:

- **Generalizing across pages**: websites commonly serve thousands of pages, and it is impractical to require a server to train a policy generation model for each page that it wishes to accelerate. However, failing to incorporate different pages during training may hide push/preload insights, and result in poorly generalizable models. To overcome this, Alohamora leverages our observation that even though sites serve thousands of URLs, their pages typically cover a far smaller number of page structures, e.g., pages are often auto-generated from fixed templates [26]. The key idea is that these *shared structural properties typically dictate the efficacy of different push/preload strategies*. Thus, Alohamora needs not train on multiple pages with the same structural properties, as those would contribute similar push/preload insights. More specifically, push/preload benefits are dictated by how browsers utilize network resources, which in turn can be characterized by 1) browser execution and inter-object dependencies, and 2) network bandwidth contention across concurrently downloading objects. By extracting this information from a site's pages and clustering pages accordingly, we find that Alohamora is able to strike a desirable balance between the number of pages required for training, and model generalizability.

- **Simulating page loads**: in order to train its policy generation model, Alohamora learns through experience, loading pages with different push/preload policies in diverse execution environments. However, the large number of potential environments and push/preload policies that exist for a given page, coupled with the high mobile page load times described above, make this approach far too resource-intensive and slow. For example, even for a single execution environment, exploring the thousands of potential policies for `nytimes.com` would require 30 days on a powerful desktop machine. To handle this, Alohamora introduces a novel page load simulator which is able to accelerate page loads by 3-4 orders of magnitude compared to running a real browser, while delivering highly faithful predicted load times with median errors of only 0.4-2.2%; for context, this results in only 20 minutes of training time for `nytimes.com` across many environments. To the best of our knowledge, Alohamora's simulator is the first to faithfully predict performance of a page *across metrics and environmental conditions* [27], without requiring profiles directly from those settings. The key insight is in *judiciously extracting invariants about the page load process and superimposing variable resource constraints*; invariants (e.g., page and browser dependencies) are collected via a single profiling run with a real browser, while variable properties about the target execution environment (network, CPU, cache, QoE metric) and push/preload policy are taken as input. Further, we note that the simulator is sufficiently general to support a wide range of optimizations (beyond push/preload) that modulate network/compute delays [2, 28, 12, 14] or scheduling across resources [10, 11].

We evaluated Alohamora using 500 web pages, a wide range of mobile network conditions, and numerous client device and cache settings. Our experiments reveal that Alohamora reduces page load times and Speed Index by 19-57% and 12-34%, respectively, compared to a default browser and standard push/preload-all policy. In addition, Alohamora yields performance similar to WatchTower [2], a recent proxy-based accelerator, and outperforms Vroom [1], a state-of-the-art server push system that employs static push/preload policies, by 2-3x. Importantly, whereas

Vroom slows down 24-34% of page loads, Alohamora's push/preload policies *never* degrade load times. We will open-source Alohamora post publication.

## 2 Background and Motivation

We begin with an overview of HTTP/2 (§2.1), and then present measurements that illustrate the potential benefits and challenges with HTTP/2's push and preload features (§2.2).

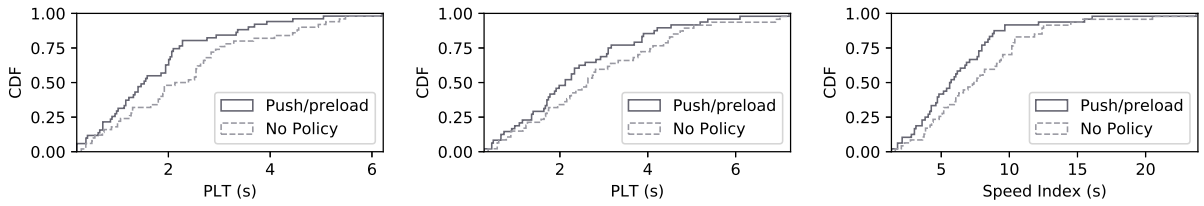
### 2.1 HTTP/2 Overview

HTTP/2 [18] alters the traditional HTTP/1.1 page load process by adding the following new features:

- **Request multiplexing:** With HTTP/1.1, browsers can open and reuse multiple concurrent TCP connections per origin. In contrast, HTTP/2 permits only a single TCP connection per origin, and allows browsers to multiplex requests onto that connection as parallel *streams*. Unlike with HTTP/1.1 pipelining, HTTP/2's multiplexing permits out-of-request-order delivery to alleviate head of line blocking.

- **Server push:** Unlike HTTP/1.1 servers which only serve objects in response to explicit client requests, HTTP/2 servers can *push* objects that they own in anticipation of future client requests. Servers have complete flexibility in defining a *push policy*, which specifies the mapping between objects that are explicitly requested and the set of files pushed along with them. Pushed resources are usable for the duration of the current page load, regardless of the associated HTTP caching headers. Note that any pushed objects that are already in the client's browser cache imply wasted network bandwidth.

- **Preload:** HTTP/2 also carried over HTTP/1.1's preload feature, which enables servers to list URLs to fetch directly in HTTP Link headers. Upon parsing such Link headers (i.e., before parsing the response body), browsers will immediately issue requests for the listed URLs; responses are not evaluated until they are referenced by the page. Thus, like push,




(a) 24 Mbps, 20 ms, cold, 1× (b) 12 Mbps, 100 ms, cold, 2× (c) 12 Mbps, 100 ms, cold, 1×

Figure 2: Push/preload benefits when policies are explicitly tuned to the available resources. Environments are listed as {bandwidth, latency, cache setting, CPU slowdown}.

preload enables servers to help browsers pre-warm their caches rather than relying on object execution to discover downstream resources. However, preload differs from push in that: 1) requests are client-driven and still involve network delays *to* origin servers, 2) the risk of re-downloading cached objects is eliminated since preload requests pass through the browser cache, and 3) servers can preload third-party objects, not just objects that they own.

- **Stream prioritization:** HTTP/2 offers a mechanism with which both clients and servers can explicitly specify how parallel request streams on a single TCP connection share network and server-side processing resources. In particular, endpoints can annotate each request with a single integer that denotes its target share of the aforementioned resources.

- **Additional modifications:** Whereas HTTP/1.1 permits browsers to download both HTTP and HTTPS resources, HTTP/2 mandates the use of TLS (and thus, HTTPS). In addition, HTTP/2 compresses HTTP headers to prevent redundant data transfers on each connection.

In this paper, we focus on HTTP/2 push/preload because they are configured by servers (Alohamora's target deployment location). In contrast, stream priorities are most often specified by browsers [29, 30], and have witnessed limited benefits [31].

### 2.2 Limitations of static push/preload policies

HTTP/2 push and preload policies have been widely studied, yielding mixed performance results [32, 23, 22, 1, 21]. The key reason is that the performance of a given policy depends on

numerous page and environmental properties. To better understand the relationships between these properties and push/preload policies, we performed a study involving 50 random pages from the Alexa top 500 US sites [19]. Our results use the same methodology and environmental parameters (network, device CPU, cache, QoE metrics) described in §6.1.

For every combination of environmental resource parameters, and for each page, we selected the best observed push/preload policy using a brute force search. Since the space of policies to consider for a page scales exponentially with the number of objects (which regular exceeds 100), a complete brute force search across environmental settings is impractical. Instead, to ensure practicality and sufficient coverage, we weighted object types based on their potential for blocking the client-side page load (i.e., JS = CSS ¿ image ¿ font) [10, 33]. To generate a policy, we randomly selected the number $N$ of objects to push/preload, and then sampled the object types $N$ times according to their relative priorities (picking randomly within each type). Finally, we randomly selected the fraction of objects to mark as push vs. preload, and for each object, we randomly selected an earlier object in the load to push/preload from. Using this approach, we generated 200 policies per page.

**Takeaway 1: Push/preload has potential.** In each environmental setting, and for each page, we compared the best push/preload policy (selected explicitly for that setting) to a default browser (i.e., no push or preload). Figure 2 shows representative results for several settings. As shown, when selected explicitly based on the environmental setting, push/preload is able to provide significant speedups over a default browser. For instance, in the {24 Mbps, 20 ms RTT, cold cache, 1x CPU slowdown, PLT} setting, median and 95th percentile benefits with push/preload are 18% and 44%, respectively.

**Takeaway 2: Push/preload policies do not generalize well.** Despite the potential benefits, our results also highlight that push/preload policies quickly degrade in performance when run outside of the precise environments for which they were tuned. To evaluate this, we performed multiple experiments in which we started with a fixed environmental setting, and selectively modulated

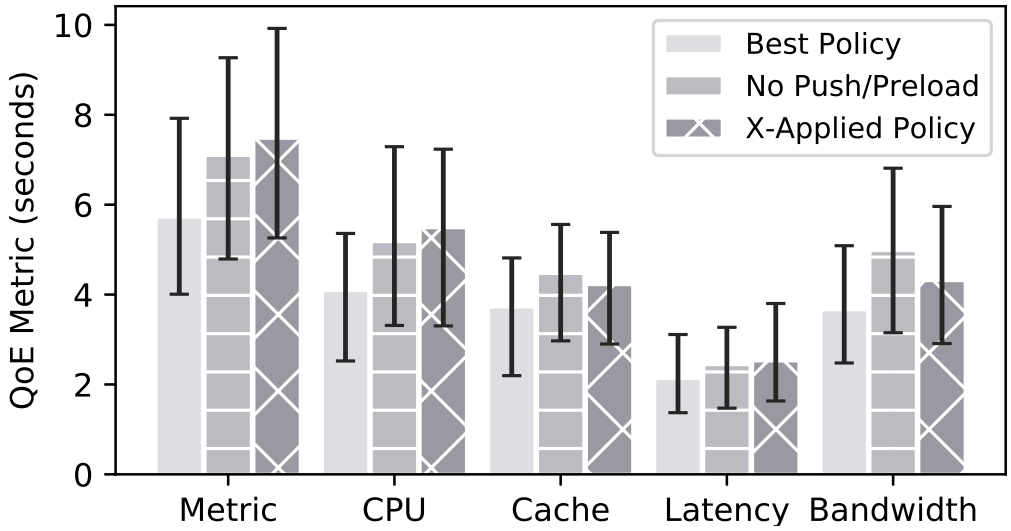



Figure 3: Push/preload performance degrades as the environment changes. The base configuration was {12 Mbps, 100 ms, cold cache, 1x CPU, PLT}; each cluster modulates only one factor. "Best Policy" was tuned to each setting, and "X-Applied" applies the base configuration's best policy to each setting. Bars show medians, with error bars spanning 25-75th percentiles.

each environmental factor while keeping the others fixed. In each resulting setting, we compared the performance of 1) the best push/preload policy from the fixed setting, 2) the best push/preload policy for the modulated setting, and 3) no push/preload (i.e., a default browser). Figure 3 depicts our results for the fixed condition, {12 Mbps, 100 ms RTT, cold cache, 1x CPU slowdown, PLT}; we omit results for other fixed conditions due to space constraints, but note that the trends persist. The results illustrate two significant drawbacks about using push/preload policies across different settings. First, they leave significant (18.4-30.7%) performance gains on the table compared to policies designed explicitly for the deployment setting. Second, and worse, they can degrade performance compared to a default browser. For instance, performance degrades by 6% and 20% at the median and 95th percentile, respectively, when device CPU speeds change. We note that these negative properties are even more pronounced when multiple environmental parameters are modified in parallel.

**Summary:** Collectively, our results suggest that, to realize the significant performance potential of push/preload, policies must be designed to explicitly consider page properties and resource

characteristics of the target deployment environment.

# 3 Design Overview

Figure 1 shows the high-level design of Alohamora's offline training phase and online (i.e., during client page loads) inference phases. In this section, we will describe the workflow for each task in the context of a single web page. We present extensions to ensure generalization across multiple pages (§4) and practical training speeds (§5) in subsequent sections.

## 3.1 Offline training

Alohamora represents its push/preload policy generator as a neural network that is trained using Reinforcement Learning (RL). RL offers several advantages in this setting compared to more standard, supervised approaches. Most notably, it is impractical to generate, a priori, a labeled dataset involving all possible push/preload policies and environments for a page, but it is paramount to incorporate all policy-affecting scenarios. RL overcomes this by using experience of prior tested policies to dynamically guide its traversal through the large search space.

In order to train its policy generation model, Alohamora runs a training phase in which the RL agent explores a web browsing environment via a large number of offline (simulated) experiments. The training process consists of a series of *episodes*, each of which aims to evaluate a series of *actions* in a given operating environment. At the start of each episode, Alohamora first selects a random operating environment by picking values for the average link bandwidth, link latency, loss rate, mobile device CPU speed, and cache settings (i.e., time since the last load, which in turn dictates the cached objects). In addition, the agent is given access to an annotated *dependency graph* [10, 11, 33, 2] for the page which lists, for each object, information about its 1) size, 2) content type (e.g., HTML, JS), 3) ordering (timing) relative to both all other page objects and only those objects belonging to the same domain, 4) cache status, and 5) candidacy for push/preload. Candidacy reflects the fact that only resources that consistently appear in a page should be considered for push/preload; we determine candidacy in the same way as prior work [1], by loading the

page several times and extracting a list of URLs that consistently appear. Collectively, the operating environment characteristics and annotated dependency graph represent the *observation* passed to the agent.

Within an episode, Alohamora maintains a running push/preload policy to which it iteratively adds selected actions. Each action is represented as a six-tuple (type, domain, $\texttt{push}_{\texttt{obj}}$, $\texttt{push}_{\texttt{parent}}$, $\texttt{preload}_{\texttt{obj}}$, $\texttt{preload}_{\texttt{parent}}$). type lists the action to perform (push, preload, nothing); domain represents the domain whose objects to consider if the action is "push"; $\texttt{push}_{\texttt{obj}}$/ $\texttt{preload}_{\texttt{parent}}$ and $\texttt{push}_{\texttt{parent}}$/ $\texttt{preload}_{\texttt{parent}}$ list the object to push/preload and the object to do so with, respectively. Key to this representation is domain, which captures the fact that push and preload actions have a different set of objects to choose from, i.e., objects can only be pushed within a domain, but preload decisions can inherently cross domains. Recall that, for objects, we have two sets of identifiers: order across all page objects, and order across objects from the same domain. For type values of "preload", Alohamora uses IDs for the page-level ordering, while for type values of "push", Alohamora uses IDs for the intra-domain ordering.

**Action space:** Throughout an episode, the agent selects actions according to a probability distribution over the potential space of 6-tuples; the probability distribution function starts as uniform, but is dynamically updated based on the agent's experiences. Each selected action is added to the running push/preload policy, and the updated policy is evaluated to obtain a reward (described below) that is fed back to the agent along with the *observation*. Each episode ends when the agent either chooses an action of type "nothing", repeats an action to push/preload an object that is already represented in the running policy, or selects an invalid (i.e., disallowed) action, e.g., pushing across domains or preloading an earlier object. Regardless of which reason ends an episode, upon completion, Alohamora automatically assigns a reward of 0 to signal to the agent that the terminal policy is not one to consider.

**Reward function:** For each action, we must evaluate the corresponding running policy to compute a reward, or the resulting performance. Note that the primary goal of the RL agent is

to maximize the expected cumulative (discounted) reward that it receives from the environment. Thus, the reward is set to reflect the performance of each tested policy according to the target QoE metric. However, structuring the reward function requires careful thought. The reason is that each action in an episode is not entirely independent. Thus, rather than simply using the page load metric of choice, we structured our reward function to take into account the relative improvement or degradation (on the metric of choice) of the policy from action to action, giving a boost in reward as the agent discovers a set of actions that leads to a new global (i.e., within the policy) minimum. More formally, we define the reward for the $i^{\text{th}}$ action in an episode as:

$$R_i(P_i, P_{i-1}, P_{\text{best}}) = \begin{cases} \frac{k_1}{P_i} & P_i < P_{\text{best}} \\ \frac{k_2 P_{i-1}}{P_i} & P_i < P_{i-1} \\ \frac{-k_2 P_i}{P_{i-1}} & P_i > P_{i-1} \end{cases}$$

where $P_i$, $P_{i-1}$, and $P_{\text{best}}$ are the raw values for the target QoE metric for the current, previous, and best-so-far policies in the episode, respectively. $k_1$ and $k_2$ are constants, where $k_1 >> k_2$. We note that the reward function is compatible with any QoE metric which denotes improved performance with lower values. We consider different reward structures in §6.4.

**Implementation:** Alohamora trains its models with Ray [34], using the RLLib [35] and Tune [36] libraries. Each model is a recurrent neural network that consists of 2 densely-connected layers with 256 units and the `tanh` activation function, followed by an LSTM with cell size 256. LSTMs are helpful given the sequential nature of each episode, whereby actions are continually added to a running policy; LSTMs prevent the agent from infinitely deferring its reward and choosing longer policies over shorter ones. Training stops after 150 iterations, or if the standard deviation in the past 50 rewards is less than 5% of the last reward (whichever comes first). Our current implementation uses the state-of-the-art A3C [37] algorithm, but is compatible with other algorithms [38, 39].

### 3.2 Online inference

As shown in Figure 1, at runtime, Alohamora introduces a frontend server (or equivalently, a reverse proxy) that is colocated with the existing backend architecture for a given domain; colocating these components ensures that end-to-end web and HTTPS security are preserved. During client page loads, all browser-generated HTTP(S) requests first hit the Alohamora frontend server, whose goals are to 1) collect the information required to query its trained push/preload policy generation model, and 2) apply the suggested policy to the current load. Each origin in a page independently runs an Alohamora server.

**Data collection for inference:** The information required to query the policy generation model matches the *observation* state used during training, i.e., average bandwidth, latency, loss rate, CPU speed, cache status, and annotated dependency graph. Alohamora collects the required network, device, and browser cache information through its interactions with clients, and the annotated dependency graphs directly from origin servers. Importantly, all of the data collection involving clients leverages existing interfaces that modern browsers expose, i.e., *Alohamora does not require browser modifications*.

To extract network latencies, Alohamora's server analyzes the SYN/SYN-ACK time during the client's initial connection setup. Further, summaries of the client's cache are collected using either the latest cache manifest standards [40], or a server-based cookie which logs the time since the user's last load of the page [41]; cache information is collected on a per-origin basis. CPU speeds are set based on the HTTP User-Agent header that denotes the client device [42]. Lastly, average network bandwidth and loss information are collected using browser user experience reports [43].

Alohamora also requires an up-to-date graph to ensure that Alohamora's server knows the precise URLs to push/preload in the current page load according to the generated policy; consequently, this graph, unlike with training, lists URLs as well as object ID numbers. Alohamora relies on origin web servers to collect and share updated dependency graphs offline, as those servers are the first to be aware of page changes. In particular, content management systems [24, 25] support

hooks that fire any time a page-altering change is pushed, e.g., for A/B testing. Alohamora adds a hook to collect up-to-date dependency graphs, which requires only a lightweight (headless) load of the largely local page [2, 12].

**Apply push/preload policies:** Upon receiving a client request for a page, Alohamora's server queries its trained model to generate a push/preload policy that directly targets the current load. The resulting policy is a listing of object IDs to push or preload, and the corresponding parents. Alohamora then uses the server-provided dependency graph to translate IDs in this policy to precise URLs. Finally, to enforce the policy, the Alohamora server issues local HTTP(S) requests (mimicking client HTTP headers) to the origin server, which responds with the up-to-date objects; Alohamora applies the generated policy to the returned object headers throughout the rest of the page load.

## 4 Generalizing Across Pages

Section 3 describes Alohamora's workflow for accelerating loads of a single page. However, in practice, sites commonly serve hundreds, if not thousands, of different pages. Unfortunately, training a separate policy generation model for each page that a website serves would be far too slow and resource intensive. Consequently, Alohamora faces a tricky tradeoff: train on only a few of a site's pages and achieve efficient training at the risk of omitting pages that warrant unique push/preload strategies, or train on many of a site's pages to develop generalizable policies at the expense of high training overheads.

Alohamora addresses this tradeoff by leveraging the observation that, even though sites serve thousands of different pages, those pages typically cover a small number of page *structures*, e.g., because they are automatically generated using a fixed set of templates, and thus share styles, layouts, and JavaScript libraries [26]. For example, news sites intuitively comprise a main home page, category home pages, and several classes of article pages. The key idea here (validated below) is that these shared structural properties typically dictate the efficacy of different push/preload strategies, and thus, we need not train on multiple pages that share structural properties.

The primary challenge associated with leveraging this observation is in determining precisely which pages in a site are necessary to consider during training. Answering this question implicitly requires an understanding of what pages have sufficient structural similarity from the perspective of the push/preload policies that they warrant. In other words, how should we represent and compare pages to determine how structurally similar they are? Our goal is for page representations to be sufficiently coarse and high-level to avoid deeming all pages as structurally different (thereby eliminating training efficiency savings), but also detailed enough to avoid classifying all pages as structurally equivalent (thereby hiding structural differences that would affect push/preload policies).

**Clustering by page structure:** In order to identify page structures that are amenable to similar push/preload strategies, Alohamora analyzes an altered version of the annotated dependency graphs described in §3. These graphs are directed acyclic graphs that capture inter-object initiator relationships, or which parent's computation triggered the fetch of a subsequent child object, as well as object sizes and content types. The reason for including only these properties is that the benefits of push/preload policies are inherently dictated by how browsers utilize their network resources to fetch content—these are the delays that push/preload strategies aim to alleviate. Such network delays are, in turn, specified by 1) browser execution models and their relation to page composition [33, 44], 2) inter-object content type dependencies, e.g., JavaScript execution blocking HTML parsing to discover downstream objects [10], and 3) network bandwidth contention across parallelized object fetches. The aforementioned dependency graphs' inclusion of page structure, content type, and object sizes address each factor, respectively.

Given these dependency graphs (or trees), Alohamora defines the distance between two page's trees $T_i$ and $T_j$ as the tree edit distance between them, where the cost of inserting/deleted a node is set to 1, and the cost of each change to either part of a node's label (content type or size) is set to 0.25. After computing the distances between each pair of trees, we construct a *distance matrix D* where $D_{i,j} = \text{distance}(T_i, T_j)$. With this, Alohamora can run any clustering algorithm that operates

| Site | Pages per cluster | Avg.cluster PLT in sec (std.dev.) | Avg. Push/preload % Improvement |
|---|---|---|---|
| **The Atlantic** | 21,5,3,1 | 1.8, 4.8, 2.7, 6.5<br>(0.1), (0.6), (0.3), (0) | **per-page**: 17, 20, 15, 34<br>**X-applied**: 78, 72, 73, 100 |
| **NPR** | 19,8,3 | 1.9, 1.6, 3.6<br>(0.3), (0.2), (0.5) | **per-page**: 15, 18, 23<br>**X-applied**: 70, 68, 75 |
| **CNN** | 17,12,1 | 11.4, 9.1, 8.6<br>(1.5), (0.5), (0) | **per-page**: 4, 3, 6<br>**X-applied**: 81, 76, 100 |

Table 1: Alohamora's cross-page generalization approach. Results are for the {24 Mbps, 20 ms, 2× CPU slowdown} setting, and consider 30 pages per site. "Per-page" results use the best policy per page in a cluster, while "X-applied" applies a single policy to all pages in each cluster.

on non-Euclidean distance functions – our implementation uses agglomerative clustering [45] – to group pages that are structurally similar from a push/preload perspective.

**Case studies:** We performed case studies on ten randomly selected websites in the Alexa top 500 US sites [19]. For each site, we ran a monkey crawler [46] that generated a list of 300 URLs by performing random interactions (e.g., clicks) starting from the site's landing page. From this set of URLs, we arbitrarily selected 30 pages that fit into the logical clusters that we perceived for the page, e.g., articles vs. home page vs. user profile pages. For each of the 30 pages, we generated the corresponding annotated dependency graph, computed the pair-wise distance metrics to all other pages, and performed the clustering.

For each cluster, we analyzed how comparable the constituent pages are along three axes: 1) general page load performance, 2) potential push/preload benefits, computed by running a brute force search to find the best policy per page (§2.2), and 3) fraction of potential benefits achieved by applying the best policy from another page in the cluster. We note that push/preload policies were easily extended across pages in a cluster given the shared structural similarities (recall that policies identify objects by IDs, not precise URLs).

Table 1 lists our results for three representative sites and a single execution environment; we note that trends generalized to the other tested conditions and sites. There are three key takeaways.

First, the generated clusters largely matched our high-level clustering intuition used when picking the pages, e.g., for The Atlantic's website, there exists a cluster for the home page, articles (21), category pages (5), and user profile pages (3). Second, the pages within a cluster exhibited highly comparable (within 15%) PLTs, both without push/preload, and with the best push/preload policy per page, and PLTs were largely separated across clusters. Lastly, and most importantly, the push/preload policies generated for any page in a cluster is able to achieve the majority (65%–100%) of potential push/preload benefits for all other pages in that cluster.

**Handling page changes:** Recall that origin servers track changes to their page dependency graphs and share those graphs with Alohamora's runtime server (§3). A natural question is how to determine when a change to a page's dependency graph is substantial enough to deem Alohamora's model suboptimal (for that page) and prompt a retrain? To answer this question, upon receiving a dependency graph from an origin server, Alohamora re-clusters by computing the pairwise distances between the new graph and all existing ones across all pages used for training. If the new clustering results remain stable such that the new graph falls into an existing cluster, then Alohamora needs not retrain. On the other hand, if the new page forms an island (i.e., a cluster of size 1), then Alohamora will automatically trigger a re-train. During re-training, Alohamora will still use its model to service applicable pages whose graphs have not substantially changed.

We note that prior work has shown that page dependency graphs remain structurally similar over long time scales (e.g., weeks), with only the precise URLs changing over short periods [11, 10, 2]. Thus, we expect retraining with Alohamora to be infrequent in practice. For example, we verified that the clustering results from Table 1 are unchanged across 2 weeks.

## 5 Page Load Simulator

To accelerate training (§3), Alohamora uses a novel page load simulator that, given an annotated dependency graph for a page, a target execution environment, and a push/preload policy as input, outputs an estimated QoE (e.g., PLT, SI) value. We will start by describing the simulator's operation in the context of cold cache page loads, no push/preload, and PLT, and then relax those

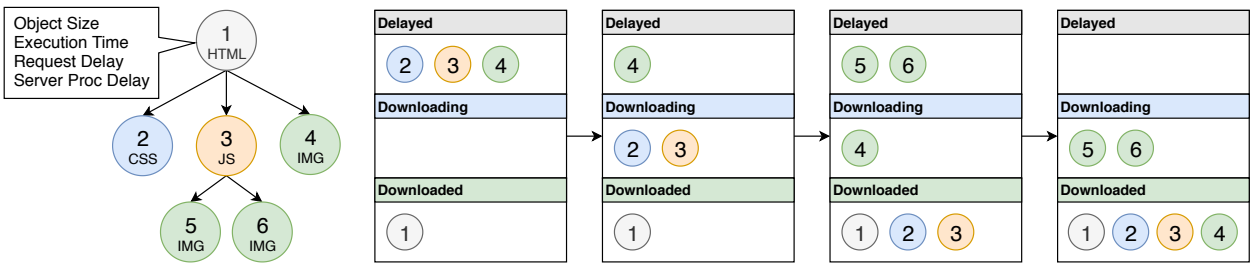



Figure 4: Example operation of Alohamora's page load simulator. After downloading the top-level HTML (omitted), the Request Queue immediately marks its children as *delayed*. Each object is *delayed* for a duration that reflects its execution time, anticipated network download delays, and inter-object dependencies (e.g., object #3 is a blocking JS file). The Request Queue steps through the *delayed* and *downloaded* queues as delays expire or objects finish downloading, and it continues to process those objects' children recursively.

assumptions. We note that Alohamora's simulator focuses on page loads that entirely use HTTP/2.

### 5.1 Collecting Simulator Inputs

The first step in the simulation process is to profile a load of the target page to extract information that characterizes properties dictated by either page composition/content [10] or browser dependencies [33, 44]. These properties do not describe the operating environment (which will we simulate), but instead dictate how page load tasks should share the simulated resources.

To extract such information, Alohamora records the target page with a record-and-replay tool [47], and replays the page over an unshaped (i.e., high bandwidth, near-0 latency) local network with desktop-level CPU resources. During replay, Alohamora extracts an annotated dependency graph (Figure 4) that matches the one used for training and inference (§3) in that it captures the inter-object ordering and dependency constraints. However, in addition, Alohamora logs per-object:

- **execution time:** the time spent parsing, executing, or rendering the object with the well-provisioned CPU; this does not include the time taken to execute any referenced objects.
- **request delay:** the amount of time between when the object's parent has finished downloading, and when the object's request is issued; this embeds the parsing/execution delays of the parent, as well as any synchronous processing delays for objects referenced earlier in the

parent's execution, e.g., a blocking external <script>.

- **server-processing delay:** unavoidable server-side delay in generating and serving the response; we extract this information directly from record-and-replay frameworks [47].

In addition to this dependency graph, Alohamora's simulator also takes as input the average network bandwidth, latency, and loss rate (Mbps, ms, %), device CPU speed (slowdown compared to profiling CPU speed), and browser cache contents.

### 5.2 Simulating the Execution Environment

In order to enforce the specified network bandwidth, latency, and device CPU values for all objects throughout the simulated page load, Alohamora's simulator uses a new **Request Queue** abstraction. In this section, we describe how the Request Queue operates on objects that have been passed into it; we describe how objects get added to the Request Queue in the next section.

At any time, the Request Queue keeps track of three types of resources: *delayed*, *downloading*, and *downloaded*. When an object is added to the Request Queue, it is set to *delayed* and is assigned a blocking delay $d$ which lists how long until it can be marked as *downloading*. $d$ incorporates the round trip latencies required to fetch the object, the profiled server processing delay, the object's *execution time*, and the *request delay* that captures blocking page load dependencies. Here we describe how the target environment affects each delay component.

**Enforcing latency/loss overheads:** In order to compute the number of round trips required to download an object, the Request Queue considers two factors. First, if the object is the first to be downloaded from a given domain, the Request Queue adds 2 RTTs to account for the TLS handshake that HTTP/2 mandates. Second, the Request Queue estimates the number of round trips required for the TCP-level data transfer by (approximately) keeping track of TCP window state for each connection (assuming cubic-like behavior) and assuming concurrent objects fairly share the window.[1] More specifically, it assumes an initial window of 10 [48], additively increases

[1] HTTP/2 stream prioritization modifies this to be weighted sharing.

the window as bytes are downloaded, and halves the window on each idle RTO (200 ms) or probabilistic lost packet. Note that this is an approximation since the currently downloading objects may complete prior to the expiration of $d$ ms, and the starting TCP window experienced may be different.

**Enforcing bandwidth overheads:** Across all objects that are concurrently marked as *downloading*, the Request Queue must enforce an appropriate split of the specified network bandwidth resources. We note that the available bandwidth (specified as a packet delivery trace [47]) is viewed as the access link, which is commonly the bottleneck in wireless networks [16], and is thus shared by the active connections across all origins. By default, the Request Queue assumes that all outstanding requests fairly share the available bandwidth, thereby disregarding discrepancies in cross-connection window state.

**Enforcing CPU overheads:** The Request Queue modulates the *execution delay* for each *delayed* object by multiplying by the slowdown magnitude; CPU speeds also affect *request delays*, which we discuss in the next section. The simulator ignores CPU core counts, and instead focuses exclusively on clock speeds, which have been shown to be the predominant factor affecting browser performance [49]. To support concurrent iframe execution, the Request Queue subtracts out execution times from concurrently *delayed* objects across frames.

**Request Queue operation:** The Request Queue proceeds in discrete "steps". In each step, the Request Queue inspects both the lists of *downloading* and *delayed* objects, and finds the object(s) that are scheduled to either finish downloading first (fewest bytes remaining) or transition to *downloading* (smallest $d$ remaining), respectively. Each step is clocked by the duration $t$ until those object events complete. After computing $t$, the Request Queue will subtract the number of bytes that can be downloaded in $t$ from all currently *downloading* objects, and subtract $t$ from all currently *delayed* objects. It will then move all *delayed* objects whose $d$ values have expired to the *downloading* queue, and mark all objects that complete *downloading* as *downloaded*.

### 5.3 Simulating Page Loads

Starting from the root node in the graph (i.e., the top-level HTML), each time an object is returned from the Request Queue as *downloaded*, the Alohamora simulator immediately adds all of that object's direct children as *delayed* to the Request Queue. In other words, each child of the completed parent object is scheduled in a one-step look-ahead process, resulting in a dependency graph traversal that is breadth-first across each objects children, but not necessarily across objects in the same row but with different parents (Figure 4).

This simple approach closely mimics the browser graph traversal strategy [10, 33], but with one issue: execution dependencies between an object's children. For instance, consider a simple scenario in which the top-level HTML includes two adjacent HTML `<script>` tags that reference files $S\_1$ and $S\_2$, both of which are the root nodes for downstream subtrees. Because browsers are unaware of the potential state dependencies between these two JavaScript files, upon discovering the first `<script>` tag, HTML parsing would halt and trigger a synchronous (i.e., blocking) fetch and execution of $S\_1$ [10]. This has several implications on dependency graph traversal, which Alohamora's simulator must account for:

- during a real page load, the children of a given parent may not be scheduled in a single burst. Alohamora's simulator accounts for this with the duration for which the Request Queue marks an object as *delayed* ($d$). In particular, recall from §5.2 that $d$ includes each object's *request delay*, which accounts for inter-children blocking delays.

- even with the enforced *request delay*, it is possible for the Request Queue to mark $S\_2$ as *downloaded* before $S\_1$, e.g., if $S\_2$ is far smaller than $S\_1$, and the simulated network is highly bandwidth-constrained. This could result in cascading discrepancies in graph traversal: $S\_1$'s children should be handled before $S\_2$'s. To handle this, Alohamora's simulator also exposes the Request Queue to the object fetch ordering that was logged during the profiled page load. These IDs closely follow the order in which browsers require, or are blocked on, specific objects because objects are fetched only after blocking constraints are satisfied.

With this information, the Request Queue treats *downloaded* objects as a priority queue, signaling object completion to the graph traversal component only once the next required object (i.e., the lowest incomplete ID) is complete. Asynchronously-fetched objects are returned after their closest synchronous neighbors.

Even with these strategies, the simulator's dependency graph traversal still faces potential inaccuracy in the fact that objects that involve a blocking dependency, such as $S\_1$ and $S\_2$ in the above example, may download concurrently and share network resources. However, the simulator bounds the cascading effects of these inaccuracies on the page load process by ensuring that the ordering of downstream children processing is unaffected and faithfully mimics that with a real browser.

**Measuring PLT:** As objects are returned from the Request Queue, they are marked with a completion time (relative to the start of the page load) accounting for any delayed or downloading time. PLT is the maximum object completion time.

### 5.4 Simulating Push/Preload Policies

To support push/preload, when an object is being added to the Request Queue, the simulator also schedules the corresponding objects to push and preload along with that object (as per the input policy). The objects added for push/preload largely share the delay value ($d$) of the parent since push/preload objects cannot begin downloading until the parent does, which in turn requires the Request Queue to impose the parent's *request delay*. However, the push/preload objects' delays are altered in two ways: 1) their server-side processing delays are preserved (and not adopted from the parent), and 2) preload objects incur an additional network RTT to account for the download of parent response HTTP headers (0.5 RTT) and transmission of the preloaded object's request (0.5 RTT).

Once scheduled, the key challenge is in determining how pushed/preloaded objects affect the delays from the profiling stage's default load; this delta could be positive or negative due to, e.g., bandwidth contention. To understand this, once the simulator hits an object that was

pushed/preloaded, it determines how the pushed/preloaded object's download progress has (or will) compare to the case when the object was not pushed/preloaded. This comparison is made by simulating the load without that resource being pushed/preloaded, and comparing the resulting delays. We note that, if the pushed/preloaded object is blocking, delays for downstream siblings are edited to reflect the observed deltas.

### 5.5 Extensions

**Additional performance metrics:** We extended the simulator to return the above-the-fold time [50], which is the time-instant version of Google's Speed Index metric (§6.1). For this, during profiling, the simulator determines which set of page objects affect the visual aspects of the browser viewport [50]. With this information, the simulator summarizes performance as the time when the last node in the collected set completes. The simulator is also amenable to Speed Index or Ready Index [50]: the profiling step must also measure the fraction of the viewport that is visually or functionally affected by each object's execution, respectively, and the simulator would track the weighted progression.

**Warm cache page loads:** In order to handle warm-cache browsing scenarios, the simulator takes an additional input: the list of resources that it should consider as cached, which can be computed by analyzing HTTP headers according to a desired warm cache timing, i.e., the time between the cold and warm cache page load [51]. The simulator then operates as normal, but sets the network RTTs required to fetch a cached resource, and the bytes that must be downloaded, to 0; *request delays* for downstream children of blocking resource are also updated.

### 5.6 Evaluations

**Faithfulness:** Figure 5 shows that Alohamora's simulator reports highly faithful load times compared to a real browser. For example, in an environment with no network or CPU shaping and a cold browser cache, the simulator's reported load times were within 0.4% and 4.3% of the real browser, at the median and 95th percentile, respectively. Median discrepancies marginally increase to 1.4% and 2.2% as network shaping and caching are incrementally added.

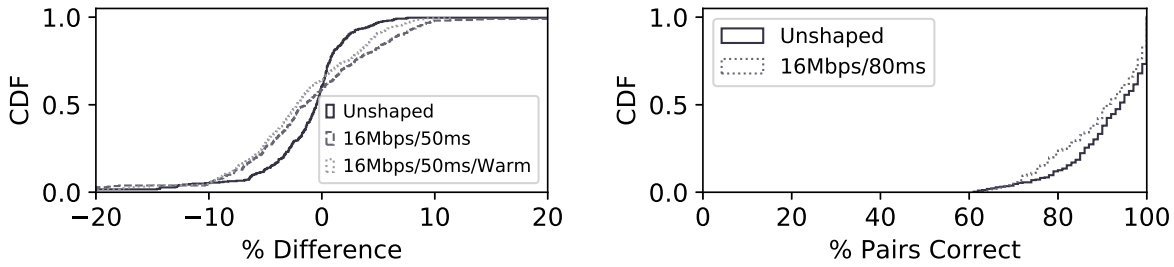



Figure 5: Faithfulness of Alohamora's simulator. (left) compares predicted PLTs with a real browser, and without push/preload. (right) shows Alohamora's ability to correctly compare push policy pairs (in terms of relative performance).

| | **Median** | **95$^{th}$ Percentile** |
|---|---|---|
| **Alohamora's simulator** | 4.7 | 22 |
| **Unshaped** | 1347 | 3815 |
| **24Mbps/20ms/2x CPU** | 5936 | 16683 |
| **12Mbps/60ms/4x CPU** | 9631 | 27765 |

Table 2: Per-page runtimes (ms) of Alohamora's simulator (top row) and a default browser in different execution environments.

With respect to push/preload policies for Alohamora, the key property required for training is to be able to determine which of two policies results in superior performance. To evaluate the simulator's faithfulness for this, we generated 20 random push/preload policies for each page in our corpus and compared the simulated and real-browser load times. As shown in Figure 5, the simulator correctly reported the relative comparisons across pairs 90% of the time.

**Runtime:** As shown in Table 2, the simulator is able to accelerate page loads by 3-4 orders of magnitude compared to a default browser. As expected, these discrepancies quickly increase as the target environment becomes more resource constrained. Table 3 shows that the simulator's runtime does steadily increase as the length of the push/preload policy under test grows. The reason is that, even though the simulator's approach to handling push/preload policies results in faithful load times, it requires re-simulations of the page a number of times that is quadratic with the policy length. We note, however, the resulting runtimes are still several orders of magnitude lower than default browsers, and Alohamora rarely requires investigation of policies longer than

| Policy length | 0 | 1-9 | 10-19 | 20-29 | 30-39 |
|---|---|---|---|---|---|
| **Runtime** | 4.7 (22) | 12 (73) | 45 (189) | 105 (348) | 172 (546) |

Table 3: Median (95th percentile) simulator runtimes in milliseconds with varying push/preload policy length.

20 objects (§6).

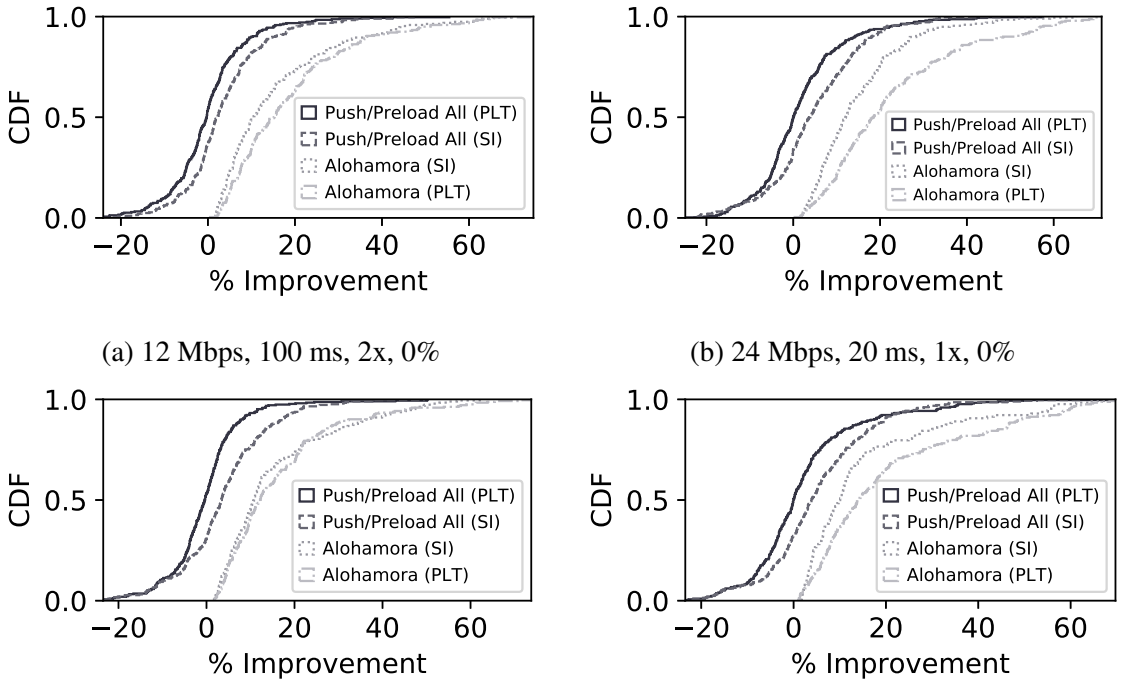
# 6 Evaluation

## 6.1 Methodology

In order to create a reproducible test environment and because Alohamora requires servers to enforce push/preload strategies, our evaluation uses the Mahimahi web record-and-replay tool [47]. Our corpus comprised the Alexa top 500 US pages [19], and we recorded versions of each page at multiple times to mimic different warm cache scenarios: back to back loads, and loads separated by 4, 12, and 24 hours. Mobile-optimized (including AMP [52]) pages were used when available. All experiments used Google Chrome for Android (v72), and all replayed page loads used HTTP/2.

Our evaluation considered a broad range of network bandwidths (6-48 Mbps, as well as Verizon and AT&T LTE traces [47]), latencies (0-100 ms), loss (0.5-5%), and client device conditions (CPU slowdowns of 1-4x, relative to a desktop with an Intel Xeon Gold 5220 CPU @ 2.20GHz). Network emulation was performed using Mahimahi [47], and CPU constraints were enforced using Chrome's Devtools Protocol [53]. Unless otherwise noted, Alohamora generated a policy generation model (across the aforementioned conditions) per page. Further, in accordance with §3, dependency graphs for inference were recorded prior to the experiments (offline).

We compared Alohamora to both a default browser, and a standard *push/preload all* strategy where, on the first incoming client request, each origin pushes all static resources that it owns, and preloads all static third-party resources that its objects reference; push/preload orders match page reference orders. Our evaluations considered multiple web performance metrics. Page load time (PLT) was measured as the time between the `navigationStart` and `onload` JavaScript




(a) 12 Mbps, 100 ms, 2x, 0%

(b) 24 Mbps, 20 ms, 1x, 0%

(c) 18 Mbps, 60 ms, 4x, 0%

(d) 18 Mbps, 60 ms, 4x, 1%

Figure 6: Load time (PLT and SI) improvements over a default browser for a static push/preload all strategy, and Alohamora. Environments are listed as {bandwidth, latency, CPU slowdown, loss rate}. Results used cold browser caches.

events. We also consider Speed Index (SI), which better relates to user-perceived performance by capturing the time needed to fully render the initial viewport. SI was measured using the pwmetrics tool [54]. Due to space constraints, we only present results for representative settings; however, we note that all presented trends persist across all tested scenarios.

### 6.2 Page Load Speedups

**Cold cache results:** Figure 6 illustrates Alohamora's ability to accelerate cold cache page loads across four representative settings. For example, in a {18 Mbps, 60 ms, 4x CPU slowdown, 0% loss} environment, median (95th percentile) PLT improvements with Alohamora were 12% (45%); the push/preload all strategy achieved only -0.2% (20%) improvements. Alohamora's benefits persist as network and CPU conditions change, although the generated policies vary (described below): benefits are 14% (60%) when 1% loss is introduced, 14% (55%) over a time-varying

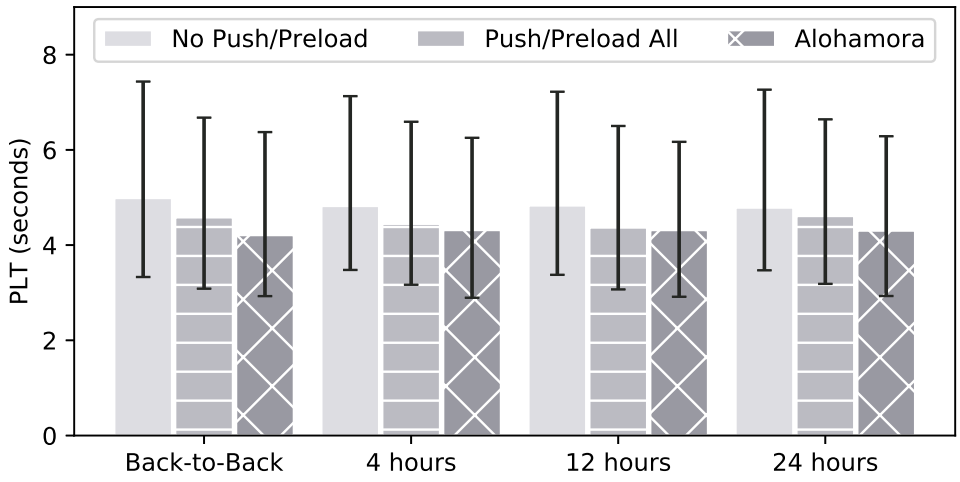


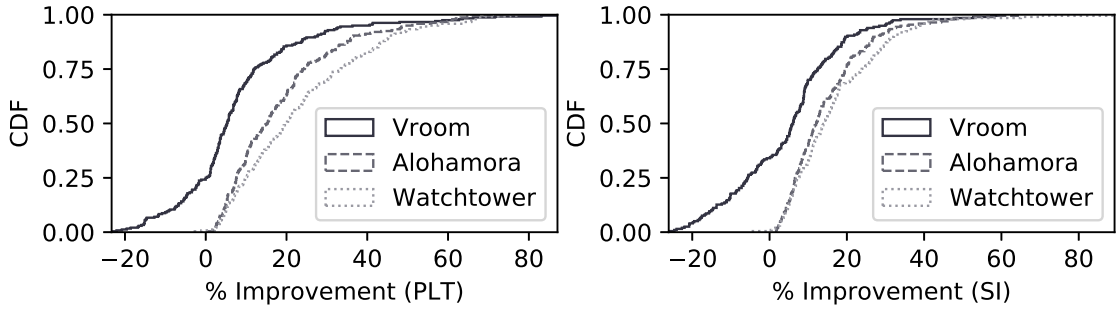

Figure 7: Load times in different warm cache scenarios; "No push/preload" is a default browser. Bars represent medians, with errors bars spanning 25-75th percentiles. Results are for the {12 Mbps, 100 ms, 2x, 0%} setting.

Verizon LTE trace (not shown), and 19% (57%) when conditions improve to {24 Mbps, 20 ms, 1x CPU slowdown, 0%}. Figure 6 also shows that Alohamora provides substantial SI benefits, ranging from 10-12% and 34-45% at the median and 95th percentile. Importantly, across all settings, Alohamora's push/preload policies *never* degraded performance compared to a default browser. This is in stark contrast to the static push/preload all policy, which, for example, slowed down 40% of pages across the settings, by up to 22%.

**Warm cache results:** Figure 7 shows that, across a wide range of warm cache browsing scenarios, Alohamora accelerates page loads compared to both a default browser and a static push/preload all strategy. For instance, for back-to-back (i.e., perfectly warm-cache) page loads, median PLT improvements are 0.8 s and 0.4 s for Alohamora and the push/preload all strategy, respectively. These relative improvements persist (10-16%) as the time between page loads increases.

### 6.3 Comparison to state-of-the-art

We compared Alohamora with two recent mobile web optimization systems, Vroom [1] and WatchTower [2]. Vroom improves upon the aforementioned push/preload all policy by using a dynamic client-side scheduler to integrate object priorities into the ordering of pushed and preloaded re-




(a) 12 Mbps, 100 ms, 2x, PLT (b) 24 Mbps, 20 ms, 1x, SI

Figure 8: Comparison with Vroom [1] and WatchTower [2].

sources. In contrast, WatchTower accelerates page loads by selectively using proxy servers that fetch objects on behalf of clients using their fast wired network links. Client-origin-proxy latencies were set as if proxies were running on EC2 in California, and WatchTower ran in HTTPS-sharding mode.

As shown in Figure 8, Alohamora is able to outperform Vroom on both the PLT and SI metrics. For example, in a {12 Mbps, 100 ms, 2x CPU slowdown, PLT} environment, benefits with Alohamora are 3x and 1.33x higher than Vroom's at the median and 95th percentile, respectively. The primary reason for this discrepancy is that, even though Vroom adds dynamism to push/preload in the form of priority-based scheduling, Vroom remains too constrained to truly adapt to diverse execution environments, i.e., the set of resources to push and preload are static and match the push/preload all approach. This is partly evidenced by the fact that Vroom still harms a large fraction of page loads, e.g., 34% in the {24 Mbps, 20 ms, 1x CPU slowdown, SI} setting. In contrast, Alohamora can vary all aspects of the push/preload policy (e.g., objects, object pairings, etc.) to best cater to the target setting. Figure 8 also shows that Alohamora is able to achieve comparable (within 15% at the median) performance to WatchTower across the two listed settings. However, we note that, unlike WatchTower, Alohamora achieves these benefits without the overheads of maintaining per-origin proxy servers.

### 6.4 Understanding Alohamora's benefits

**Ablation study:** In order to understand the relative impact of each of Alohamora's features and

| | Reward | LSTM | BW | CPU | Latency |
|---|---|---|---|---|---|
| **C1** | 61 (41) | 57 (44) | 45 (38) | 41 (36) | 66 (96) |
| **C2** | 66 (43) | 60 (47) | 58 (61) | 55 (61) | 71 (89) |

Table 4: Impact of removing features/properties in Alohamora's model. Results are reported as median (95th percentile) percentage of potential (i.e., with Alohamora's full models) improvements. "Reward" considers the intuitive $-PLT$ reward function. C1 and C2 are the {12 Mbps, 100 ms, 2x} and {24 Mbps, 20 ms, 1x} settings, respectively.

model properties, we performed an ablation study (Table 4). Our results reveal that bandwidth, latency, and CPU speed information all play a significant role in Alohamora's ability to generate performant push/preload policies, with the removal of CPU resulting in the largest degradations (45-59%). Our results also highlight the importance of Alohamora's reward function and incorporation of LSTM. For instance, (intuitively) setting the reward to $-PLT$ leads to performance degradations around 35%. The primary reason is that it becomes easy for the agent to artificially inflate the observed reward by selecting policies with fewer actions, i.e., the earlier policies in an episode will be favored as the cumulative reward will be lower. Removing LSTM, on the other hand, led to degradations of $\sim$40%, largely due to the lack of a discount factor that guides the agent to avoid unnecessarily favoring longer policies to improve the cumulative reward (§3).

**Alohamora's policies:** To understand the learned insights behind Alohamora's benefits, we analyzed its generated push/preload policies across all tested settings and pages. Admittedly, we observe that policy composition and the mix between push/preload varied dramatically across pages and resource settings; indeed, subtle interactions between these properties were a primary motivator for Alohamora's machine learning-based approach. However, we note the following common principles:

- In lower bandwidth settings, Alohamora either 1) reduced the policy length or cut data-intensive objects, or more commonly, 2) spread the same set of pushed/preloaded objects out across a larger set of parents in order to stagger downloads and reduce bandwidth contention.
- With slower CPUs, Alohamora's policies are careful to only push objects whose bytes could

be downloaded until the next blocking JavaScript file is required; the goal is to prevent downstream CPU tasks from blocking on the network. In these cases, Alohamora's policies preloaded additional resources with the goal of having their downloads start (after the 0.5 RTT required to contact the server) only after the next blocking resource was downloaded. In essence, the idea is to perfectly interleave downloads of non-blocking resources with the execution of blocking resources.

- For image-heavy sites (e.g., `pinterest.com`), Alohamora commonly excluded JavaScript/CSS files from its policies, and instead pushed/preloaded images that are rendered towards the top of the viewport, particularly in high-bandwidth settings or when SI is the target metric. The reason is that these pages have flat (not deep) dependency graphs, so blocking JavaScript files do not trigger cascaded serial network fetches; instead, image downloads have a larger blocking impact on load times.

**Inference times:** We find that Alohamora's policy generation, which occurs during client page loads, adds negligible delays to overall load times: median and 95th percentile inference times are 11 ms and 40 ms, respectively.

### 6.5 Additional results

**Incremental deployment:** Since origins make independent push/preload decisions with Alohamora, we ran experiments to understand how Alohamora's benefits vary with different adoption rates. For each page in our corpus, we ordered the domains in the page according to the fraction of objects that they contribute. We then ran experiments where only the top X% of origins used Alohamora; origins not running Alohamora did not push/preload any objects. We also specifically considered the case where only the top-level origin deployed Alohamora. As expected, Figure 9 reveals that benefits increase as more domains adopt Alohamora. However, we observe that simply having the top-level origin can achieve 56% of the potential (i.e., with 100% adoption) median benefits.

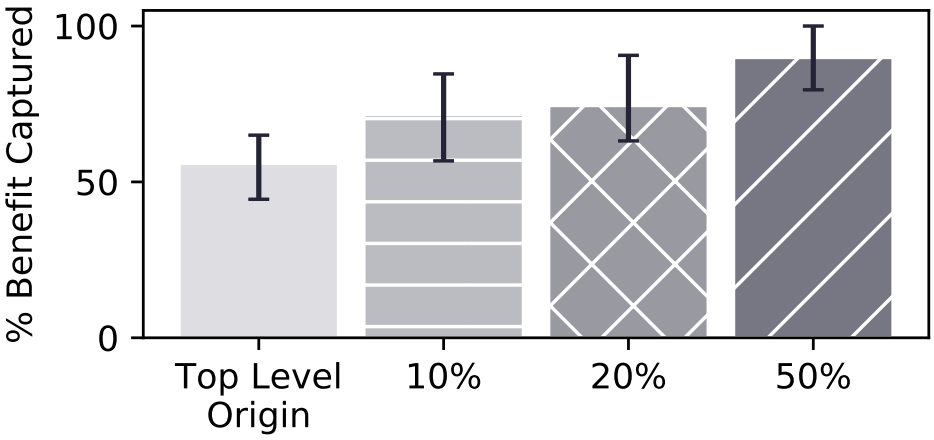



Figure 9: Percentage of potential benefits achieved when X% of origins in each page run Alohamora. Results are for the {12 Mbps, 100 ms, 2x} setting. Bars show medians, with error bars spanning 25-75 percentiles.

**Cross-page clustering:** To this point, the presented results considered Alohamora models that were trained for a single page (across environments). In order to evaluate Alohamora's ability to train generalizable models across a site's pages, we consider the three sites presented in §4 (Table 1). For each site, we trained a single Alohamora model using only a single (randomly selected) page from each cluster, and evaluated across all of the site's pages. Alohamora's cross-page models are able to achieve within 85-90% of the improvements achieved when training individually on each tested page; this slight degradation comes with the significant benefit of improved training efficiency.

**Robustness to network errors:** To generate push/preload policies, Alohamora's models ingest a variety of observations that collectively characterize the execution environment. While device CPU and cache contents require zero approximation to collect, network measurements (bandwidth and latency) can be noisy. We evaluated Alohamora's ability to deliver speedups in the face of noisy network value inputs by considering average bandwidth and latency errors of {1, 2, 3} Mbps and {10, 20, 30} ms. We find that Alohamora's generated policies are robust to such errors. For instance, median PLT improvements dropped by only 3.4% and 4.6% in the {24 Mbps, 20 ms, 2x CPU slowdown} environment with errors of 2 Mbps and 20 ms, respectively.

## 7 Related Work

**Server push systems:** Numerous studies have explored the performance of HTTP/2 (formerly SPDY), both with and without server push and preload [22, 21, 32, 23, 55, 1]. Like us, these works have found mixed performance benefits due to the subtle relationships between HTTP/2 and network characteristics, page composition, and TCP semantics. However, these prior efforts have all investigated and promoted static policies and configuration guidelines. In contrast, Alohamora leverages a data-driven approach to dynamically tune push/preload policies by explicitly factoring in both page composition and the target execution environment.

**Mobile-optimized pages:** Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [52, 56] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Alohamora accelerates legacy web pages without requiring developer effort. Further, Alohamora's adaptive push/preload policies can improve the performance of AMP pages because all page resources still must traverse a client's slow mobile access link.

Certain systems, most notably Prophecy [12], automatically rewrites web pages and returns post-processed versions of objects to clients that reduce client-side computation and network costs. Unlike Prophecy, Alohamora does not alter page content, which has proven to be highly error-prone in practice [28]. Further, we note that Alohamora can accelerate Prophecy pages which require at least one HTML file per frame, and unmodified image and style files—these are the static files which Alohamora primarily targets for push/preload.

**Proxy-based accelerators:** Compression proxies [28, 57, 58, 59] compress objects in-flight between clients and servers, while remote dependency resolution proxies [14, 2, 47, 60] perform certain object fetches and computations on behalf of clients. Though performant, such acceleration proxies violate the end-to-end security guarantees of HTTPS. WatchTower [2] addresses this dilemma, but at a significant deployment cost, as each origin in a page must operate its own proxy.

Alohamora avoids such security concerns by relying only on end-to-end HTTP/2 optimizations.

**Prefetching:** Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [61, 62, 63]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [64]. In contrast, Alohamora generates push/preload policies only after a user navigates to a page, and considers the environmental conditions and page properties collected in situ.

**Dependency-aware scheduling:** Klotski [11] analyzes pages offline to identify high-priority objects, and uses knowledge of network bandwidth and page structure to stream them to clients before they are needed. Klotski's dynamic prioritization hinges on global knowledge of object fetches, which proxies provide at the cost of security; in contrast, Alohamora origins operate independently, and leverage generalizable models which hedge against the decisions that other origins may make. Polaris [10] uses a client-side request scheduler that reorders requests to minimize the number of effective round trips in a page load without violating state dependencies. However, unlike Alohamora, Polaris relies on clients to discover page resources, and thus cannot eliminate certain serial fetches.

# 8 Conclusion

Configuring HTTP/2 push/preload policies has proven challenging, as benefits depend on complex interactions between page, network, device, and browser properties. However, to date, systems and guidelines have focused entirely on static policies that fail to generalize. This paper presents Alohamora, a mobile web optimization system that dynamically generates HTTP/2 push/preload policies using Reinforcement Learning. In order to practically realize this data-driven approach, Alohamora introduces novel techniques that drastically reduce the number of pages that must be considered for training, and the cost of training any one page—these benefits come without a drop in model generalizability or potential benefits. Across a broad range of pages, networks, and device conditions, we find that Alohamora accelerates default browsers and state-of-the-art push systems

by 19-57% and 2-3x, respectively.

## References


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

[2] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

[3] Eric Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[4] Darrell Etherington. Mobile internet use passes desktop for the first time, study finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[5] Christo Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[6] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[7] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[8] Dennis F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[9] Tammy Everts and Tim Kadlec. WPO stats. https://wpostats.com/, 2019.

[10] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[11] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha Madhyastha, and Vyas Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[12] Ravi Netravali and James Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[13] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[14] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[15] Daniel An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[16] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[17] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[18] Mike Belshe, Martin Thomson, and Roberto Peon. Hypertext transfer protocol version 2 (http/2). 2015.

[19] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[20] Torsten Zimmermann and Oliver Hohlfeld. Skip to the article Adoption, performance, and human perception of HTTP/2 Server Push. https://blog.apnic.net/2018/04/26/adoption-performance-and-human-perception-of-http-2-server-push/, 2018.

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

[22] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.

[23] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[24] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[25] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[26] Optimizely. Content Management System. https://www.optimizely.com/optimization-glossary/content-management-system/, 2019.

[27] Kyriakos Zarifis, Mark Holland, Manish Jain, Ethan Katz-Bassett, and Ramesh Govindan. Modeling http/2 speed from http/1 traces. In *International Conference on Passive and Active Network Measurement*, pages 233–247. Springer, 2016.

[28] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

[29] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1755–1764, 2018.

[30] Andy Davies. HTTP/2: Discover the Performance Impacts of Effective Prioritization. https://developer.akamai.com/blog/2019/01/31/http2-discover-performance-impacts-effective-prioritization, 2019.

[31] Patrick Meenan. HTTP/2 Prioritization. https://calendar.perfplanet.com/2018/http2-prioritization/, 2018.

[32] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and Kadangode K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, December 2015.

[33] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577. USENIX Association, 2018.

[35] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[36] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[37] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[39] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[40] K. Oku and Y. Weiss. Cache Digests for HTTP/2. https://httpwg.org/http-extensions/cache-digest.html, 2019.

[41] DeNA Co., Ltd. H2O Cache-Aware Push. https://h2o.examp1e.net/configure/http2$_d irectives.html$, 2019.

[42] Usama Naseer and Theophilus Benson. Configtron: Tackling network diversity with heterogeneous configurations, 2019.

[43] Google Chrome. Chrome User Experience Report.

[44] Javad Nejati and Aruna Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[45] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[46] Seleniumhq browser automation. https://selenium.dev/, 2019.

[47] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[48] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[49] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R. Das, and Michael Ferdman. Impact of Device Performance on Mobile Internet QoE. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 1–7. ACM, 2018.

[50] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[51] Ravi Netravali and James Mickens. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, HotMobile '18, pages 63–68. ACM, 2018.

[52] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[53] Google Developers. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/.

[54] Paul Irish. pwmetrics: Progressive web metrics. https://github.com/paulirish/pwmetrics, 2019.

[55] Utkarsh Goel, Moritz Steiner, Mike P. Wittie, Martin Flack, and Stephen Ludin. Http/2 performance in cellular networks: Poster. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 433–434. ACM, 2016.

[56] Byungjin Jun, Fabian E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[57] Shailendra Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Ramesh Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[58] Joseph Volpe. Nokia Xpress brings cloud-based compression to the Lumia line. Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[59] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[60] Ahiwan Sivakumar, Chuan Jiang, Seong Nam, P.N. Shankaranarayanan, Vijay Gopalakrishnan, Sanjay Rao, Subhabrata Sen, Mithuna Thottethodi, and T.N. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[61] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[62] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[63] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[64] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Christopher Riederer. give in to procrastination and stop prefetching.