# VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution

Vaspol Ruamviboonsuk
University of Michigan

Ravi Netravali
MIT

Muhammed Uluyol
University of Michigan

Harsha V. Madhyastha
University of Michigan

## Abstract

The existing slowness of the web on mobile devices frustrates users and hurts the revenue of website providers. Prior studies have attributed high page load times to dependencies within the page load process: network latency in fetching a resource delays its processing, which in turn delays when dependent resources can be discovered and fetched.

To securely address the impact that these dependencies have on page load times, we present VROOM, a rethink of how clients and servers interact to facilitate web page loads. Unlike existing solutions, which require clients to either trust proxy servers or discover all the resources on any page themselves, VROOM's key characteristics are that clients fetch every resource directly from the domain that hosts it but web servers aid clients in discovering resources. Input from web servers decouples a client's processing of resources from its fetching of resources, thereby enabling independent use of both the CPU and the network. As a result, VROOM reduces the median page load time by more than 5 seconds across popular News and Sports sites. To enable these benefits, our contributions lie in making web servers capable of accurately aiding clients in resource discovery and judiciously scheduling a client's receipt of resources.

## CCS Concepts

• **Information systems** → **Mobile information processing systems**; *Browsers*; • **Networks** → **Network performance evaluation**; *Network measurement*;

## Keywords

Web performance, Mobile web, Page load times

**ACM Reference format:**
Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution . In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098851


*SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA*

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098851

## 1 Introduction

Despite the rapid increase in mobile web traffic [11], page loads on mobile devices remain disappointingly slow. Recent industry studies [9, 13], as well as our own measurements (§2), show that a large fraction of mobile-optimized websites load much slower than user tolerance levels, even on state-of-the-art mobile devices and cellular networks. For example, the average web page takes 14 seconds to load even on a 4G network [12].

Since the speed of page loads critically impacts user experience and thus provider revenue [19], much effort has been expended by both industry and academia to identify the root causes for poor web performance. Recent studies [35, 41, 42] have found that dependencies between the resources on any web page are a key reason for slow page loads. Today, even mobile-optimized web pages include roughly one hundred resources [7] on average, and client browsers can discover each of these resources only after they have fetched, parsed, and executed other resources that appear earlier in the page. For instance, a browser may learn that it needs to fetch an image after executing a script which it discovers after downloading and parsing the page's HTML.

Prior work has taken one of two approaches to address the impact of these dependencies on web performance, and both approaches suffer from fundamental drawbacks.

- **Offloading to proxies.** In one class of solutions, when a client loads a page, discovery of resources on the page is offloaded to a proxy [36, 40, 43]. Solutions that take this approach attempt to reduce page load times by leveraging the faster CPUs and network connectivity of proxy servers. However, clients must trust that proxies preserve the integrity of HTTPS content; proxies that disregard HTTPS traffic are limited in the benefits they can provide given the increased usage of HTTPS [33]. Moreover, to preserve the ability of web providers to personalize content, a client must share with the proxy its cookies for all domains from which resources must be fetched to load the page.
- **Reprioritizing requests at client.** An alternative class of solutions [22, 35] lets the client itself discover all resources on a page. Instead of fetching resources in the order that they are discovered, these systems preferentially fetch certain resources (e.g., those that lead to longer dependency chains [35]) based on precomputed, high-level characterizations of the page's dependency structure. The problem with these approaches is that, once the client browser discovers the need for a resource, the client must necessarily wait for that resource to be fetched over the network before it can start processing the resource, resulting in under-utilization of the CPU. Since the client CPU is the primary bottleneck when loading web

pages on mobile devices ([34] and §2), this class of solutions has limited ability to speed up the mobile web.

These limitations of prior approaches motivate the need for a new solution that both preserves the end-to-end nature of the web and aids clients in discovering the resources on any page. Specifically, a client must receive every resource directly from the domain hosting that resource, thereby enabling the client to verify the integrity of HTTPS content and requiring the client to share its cookies for a domain only with servers in that domain. Yet, the new solution must also preserve the primary benefit of proxy-based dependency resolution, which is to decouple the client's downloading of resources on a page from the processing of those resources. Decoupling these functions maximizes resource utilization during page loads because fetching of resources is constrained by the network, whereas the CPU limits the parsing and execution of each fetched resource.

We argue that the way to realize this desired end-to-end solution is to redesign page loads such that web servers securely aid clients in resource discovery. In addition to returning a requested resource, a web server should inform the client of other dependent resources that the client will need in order to load the page. Though there are resource overheads associated with identifying these dependent resources, content providers have a strong incentive to incur this burden in order to decrease load times for their clients, thereby increasing revenue [6, 46]. We make three contributions in designing VROOM to realize this approach.

First, to aid clients in resource discovery, VROOM-compliant web servers not only *push* the content of dependent resources (leveraging the server push capability in HTTP/2 [20]) but also return *dependency hints* in the form of URLs for resources that the client should fetch. The use of HTTP/2 push alone is insufficient because content on modern web pages is often served by multiple domains [21], each of which can only securely push the content that it owns. In contrast, dependency hints enable a server to inform a client of the dependent resources that it should fetch from *other domains*, without providing the content of those resources. This additional input from servers ensures that a client's ability to discover and start downloading required resources is not constrained by the speed with which it can process fetched content. In fact, by the time the client discovers the need for a resource during its execution of a page load, that resource will likely already be in its cache.

Second, we develop the mechanisms that VROOM-compliant web servers must employ to identify the resources they should push and the dependency hints they should include with their responses. In contrast to prior efforts, which have relied exclusively on either online [36, 40] or offline [22, 35] dependency resolution, we show how to *combine* the two approaches to accurately identify the set of resources that a client will need to fetch within a specific page load. Critically, our design maximizes the number of dependent resources that the client is made aware of while avoiding sending dependency hints for intrinsically unpredictable resources—ones which vary even across back-to-back loads of the page—so that the client does not incur the overhead of fetching resources that are unnecessary for its page load.

Lastly, while the additional input from VROOM-compliant web servers reduces the latency for clients to discover all resources on a web page, fetching all resources as soon as they are discovered

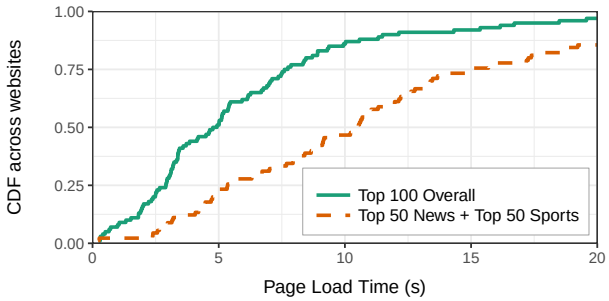



**Figure 1: Page load times on today's mobile web.**

increases contention for the access link's bandwidth, delaying the receipt of some resources. To maximize CPU utilization, we leverage the property that resources that need to be parsed/executed (HTML, CSS, and JS objects) constitute only a quarter of the bytes on the average mobile web page [7]. We coordinate server-side pushes and client-side fetches such that resources that need to be parsed/executed are received earlier than other resources; the client can fetch the latter set of resources while processing the former set. In doing so, we ensure that resources arrive at the client in the order in which they will be processed.

Our implementation of VROOM enables the use of Google Chrome to load pages from the Mahimahi [36] page load replay environment. On a corpus of web pages from popular News and Sports sites, the median page load time reduces from the status quo of 10.5 seconds to 5.1 seconds with VROOM. These improvements stem from VROOM's ability to enable server-side identification of dependent resources with a median false negative rate of less than 5%, which in turn results in a 22% median decrease in client-side latency to discover all resources on a page.

## 2 Motivation

We begin by presenting a range of measurements that illustrate the poor web performance today on mobile devices, estimate the potential to reduce page load times, and show that existing solutions are insufficient.

**Problem: Poor load times.** We demonstrate the slowness of the mobile web using two sets of websites: the Alexa US top 100 websites and the top 50 sites each in the News and Sports categories; these popular sites apply known best practices such as minifying JavaScript content and eliminating HTTP redirects [4]. We load the landing page for each site five times on a Nexus 6 smartphone that is connected to Verizon's LTE network with excellent signal strength; we report median page load times.[1]

Figure 1 shows that the median site among the top 100 takes roughly 5 seconds to load, which is higher than the 2–3 second period that a typical user is willing to wait [27]. When considering News and Sports sites, which are more complex than the average site [21], the median load time is even higher, exceeding 10 seconds. Since the need for faster loads is particularly acute on News and Sports sites, we focus on these sites in the rest of this section.

**Cause: Poor CPU/network utilization.** We now consider how low page load times can be reduced to without web pages being rewritten [1]. For a client to not have to trust proxies to execute

[1] We compute page load time as the time between when a page load begins and when the `onload` event fires.

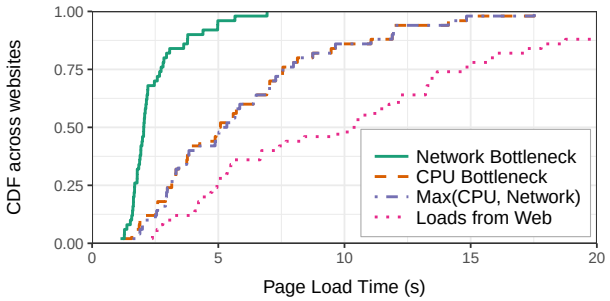



**Figure 2: Potential for reducing page load times by making better use of the client's CPU and network.**

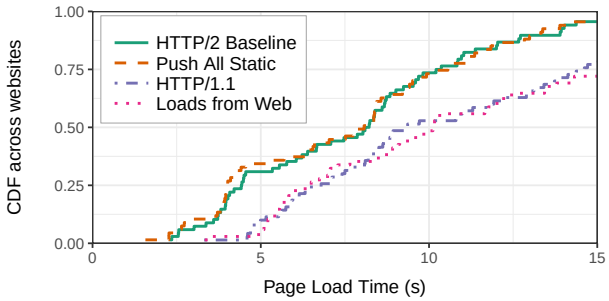



**Figure 3: Estimation of page load time improvements that would be enabled once HTTP/2 is globally adopted. Given that the adoption of HTTP/2 is still in its nascency today, the fidelity of our replay setup is confirmed by the close match between load times measured when loading pages on the web and in our HTTP/1.1 replay environment.**

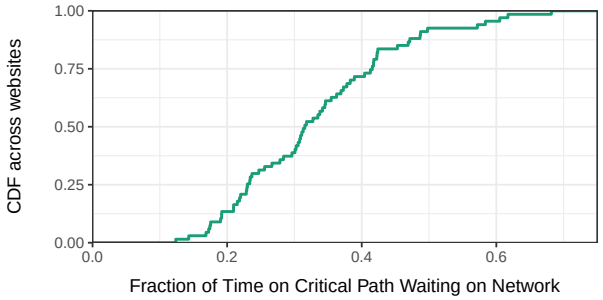



**Figure 4: Fraction of the critical path spent waiting for the network, when the client uses HTTP/2 to communicate with all domains.**

page loads on its behalf, the client must fetch all resources on a page directly from origin web servers and locally process all of these resources. This places two constraints on web performance: the client's network connection and its CPU. To compute a lower bound on page load times, we estimate the potential gains from a redesign of the page load process that would fully utilize at least one of these two resources.

To mimic a setting where the network bandwidth is the bottleneck, we replay each page load after modifying the root HTML to list all resources required to load the page in a manner that instructs the browser to fetch these resources but not evaluate them. To emulate a setting where the client's CPU is the bottleneck, we load every page with the client phone connected via USB to a desktop which hosts all of the web servers. In both cases, we use Mahimahi [36] to record page content and replay page loads, and we use HTTP/2 between the client and all web servers in order to make efficient use of the network. We use the same mobile device and cellular network as above, and we further describe our replay setup in Section 6.

Figure 2 shows that, when exactly one of the network or the CPU is the bottleneck, rather than both limiting each other as is the case today, page load times on popular News and Sports websites are significantly lower than the status quo (the median load time drops from 10.5 seconds to 5 seconds). Our results also show that the CPU is typically the bottleneck in mobile page loads, corroborating the findings of recent studies [34]. Furthermore, page load times in the case where the CPU is the bottleneck remain largely the same even if we disable 1 of the 4 cores on the Nexus 6 smartphone, indicating that adding more cores will not help improve mobile web performance.

**Existing solutions are insufficient.** Since we seek a solution that improves mobile web performance while preserving the end-to-end nature of the web, we consider two existing solutions that satisfy this property.

First, we consider a setting where all domains on the web have adopted the latest version of the HTTP protocol, HTTP/2. HTTP/2 reduces inefficiency in the use of the network by enabling requests to be multiplexed on the same TCP connection. To estimate the potential impact of HTTP/2, we replay page loads in an environment where HTTP/2 is universally used ("HTTP/2 Baseline"). The results in Figure 3 portend that, though the global adoption of HTTP/2 will reduce the median page load time across popular News and Sports websites to roughly 8 seconds, mobile web performance will remain significantly short of optimal; the lower bound we saw in Figure 2 was more than 2 seconds lower, a substantial gap given that web providers have found that even a few 100 milliseconds of additional delay significantly reduces their revenue [10]. Configuring the first party domain of every page to push (leveraging HTTP/2's server push capability) all static resources that it hosts offers little additional benefit, for reasons discussed later.

Performance with HTTP/2 falls significantly short of the lower bound because the root cause for high page load times remains: since both CPU-bound and network-bound activities are typically on the critical path of a page load [41], neither the client's CPU nor its access link is utilized to capacity. The client browser cannot parse an HTML/CSS object or execute a JavaScript file until it has incurred the latency to fetch that resource, which in turn it can begin to do only after discovering the need to fetch that resource by parsing/executing another resource. Indeed, Figure 4 shows that a significant fraction of time on the page load's critical path—over 30% on the median page—is spent waiting to receive data over the network, leading to under-utilization of the CPU, the bottleneck resource (Figure 2).

An alternative end-to-end solution for improving web performance is to use an approach like Polaris [35]. With Polaris, the client receives a characterization of the page's dependency structure at the start of the page load and uses this knowledge to prioritize requests for more critical resources. However, such an approach can do little to reduce the network delays encountered on the critical path. The fundamental constraint with this approach is that the client must discover all resources on the page on its own (i.e., fetching a resource and then evaluating it to identify new resources to fetch). As a result, once the browser discovers a resource by parsing/executing other resources that appear earlier in the page load, the latency of fetching

that resource must be incurred *at that time* before the browser can begin processing the resource. Since HTTP/2 eliminates head-of-line blocking and network bandwidth is not the bottleneck in mobile page loads (Figure 2), reordering requests for discovered resources offers little benefit. We provide further evaluation and discussion of Polaris in Section 6.

**Summary.** Together, the measurements in this section lead to the following takeaways:

- Page loads on mobile devices are currently significantly slow even for popular websites, particularly for sites in categories that have more complex web pages than others.
- The reduction in load times that we can expect from the adoption of existing end-to-end solutions will not suffice.
- However, we could potentially halve the median load time if the page load process were redesigned to more efficiently use the client's CPU and network.

## 3 Approach

Our results in the previous section indicate that the key to optimizing mobile web performance is to maximize the utilization of the client's CPU. To do this, we need to ensure that the browser's processing of any resource is not delayed waiting to receive the resource over the network. To achieve this decoupling of the browser's use of the CPU and network, web page loads would ideally work as follows. When a client browser issues a request for a page, it would receive back *all* the resources needed to render the page, rather than just the HTML for the page. This would optimize page load performance for two reasons. On the one hand, in contrast to the status quo wherein the client incrementally fetches resources as it discovers them during the page load, receiving all of the objects on the page at once would maximize the utilization of the client's access link and eliminate the need for repeated latency-onerous interactions between the client and web servers. On the other hand, if the resources on the page are delivered in the order they need to be processed, the client can make full use of its CPU, processing resources while fetching other resources in parallel.

### 3.1 Limitations of HTTP/2 PUSH

It may appear that such an ideal design of the page load process is feasible today because HTTP/2-compliant web servers can speculatively *push* resources to clients [20]. However, today, 1) the resources on a page are often spread across multiple domains [21], e.g., a web page from one provider often includes advertising, analytics, JavaScript libraries, and social networking widgets from other providers; 2) HTTPS adoption is rapidly growing [8, 33]; and 3) page content is increasingly personalized. These typical characteristics of modern web pages make the use of HTTP/2 PUSH inefficient for the following reasons.[2]

- When a domain receives a request for the HTML of a page that it hosts, it can only return resources that it hosts and not resources served by other domains. In the example in Figure 5, in response to the request for the HTML, `a.com`'s servers can only push the contents of `foo.js` which is served from the same domain, but not the third-party resource `img.jpg`. If web servers were to fetch resources from external domains and push them to clients [22, 36, 43], clients would be unable to verify the integrity of secure page content. Moreover, since any client's request to a web server will only include the client's cookie for that domain, resources fetched from other domains by that web server will not reflect any personalization of content by those domains.
- If web servers only push locally hosted resources, the client will discover resources that it needs to fetch from other domains only after processing previously fetched resources, e.g., in Figure 5(b), the client can discover the need to fetch `img.jpg` only after executing `foo.js`. This makes the CPU a potential bottleneck in the client's fetching of resources.

[2] Another commonly cited limitation of PUSH is the potential for bandwidth wastage when a resource cached at the client is pushed. However, this problem could be addressed by having the client send a summary of its cache contents to web servers, e.g., in a cookie [5].

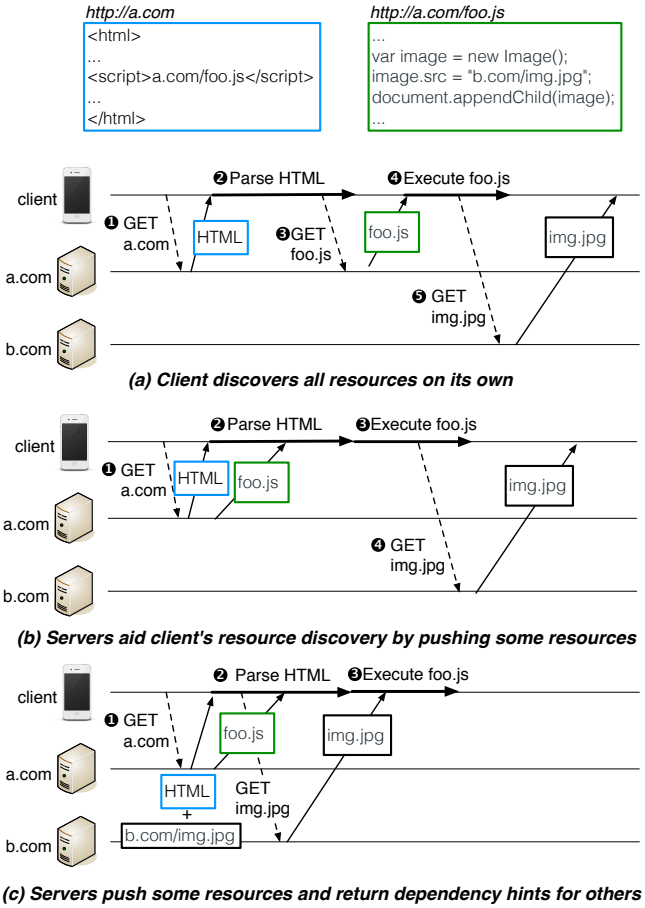



**Figure 5: Comparison of critical path across different approaches for loading web pages, in all of which the client receives every resource from the domain from which it is served, so as to preserve personalization and the client's ability to verify the integrity of secure content: (a) 5 stages on critical path with CPU use blocking use of the network in steps 2 and 4 and vice-versa in steps 3 and 5, (b) 4 stages on critical path with CPU use blocking use of the network in steps 2 and 3 and vice-versa in step 4, (c) 3 stages on critical path with CPU and network utilized throughout.**

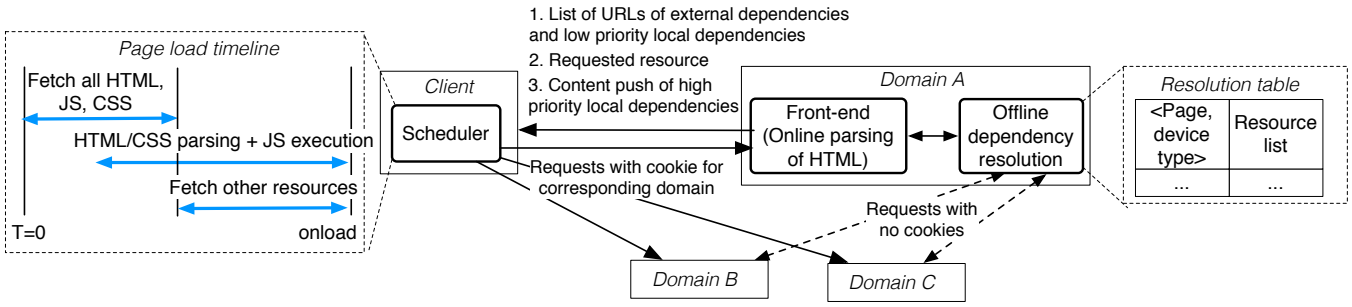



**Figure 6: Illustration of the components in VROOM and the interactions between them.**

- During a page load, if every domain independently pushes its resources to the client, the client's receipt from one domain of a resource that must be processed (i.e., HTML, CSS, or JavaScript) can be delayed due to bandwidth contention on the client's access link with other resources (such as bulky images) from other domains. This makes the network a potential bottleneck in the client's processing of resources.

### 3.2 Combining PUSH with dependency hints

Given these limitations associated with relying *solely* on HTTP/2 PUSH, we leverage an additional server-side capability. When a web server receives a request for a resource, in addition to pushing the content for some dependent resources that it owns, the server can also return a list of URLs for other dependent resources—we refer to this list as *dependency hints*. Web servers can include such a list of URLs as an additional header in HTTP responses.

When a client receives dependency hints, it can fetch every resource whose URL is included in the list, without having to first process other resources on the page to discover the URLs in the list. For example, in Figure 5(c), based on the hint received from `a.com`, the client can fetch `img.jpg` from `b.com` without having to wait to receive and execute `foo.js`.

Legacy browsers already support dependency hints in the form of HTTP Link headers which have the preload attribute set [16]; resources listed in these headers are immediately fetched by browsers but are not evaluated until they are referenced by the page (i.e., Link preload headers are primarily used to prewarm browser caches during page loads).

Using dependency hints in addition to HTTP/2 PUSH offers several advantages:

- Any web server can safely send clients dependency hints for third-party resources. For any URL received via dependency hints, a client will fetch it directly from the respective domain, enabling the client to confirm the integrity of resources served over HTTPS and preserving that domain's ability to personalize content.
- The client need not fetch resources that are already cached locally, making it easier to minimize bandwidth waste compared to the use of HTTP/2 PUSH.
- The client maintains control over its concurrent fetches of resources from multiple domains; it can coordinate its downloads such that high priority resources (those that must be processed) are not delayed.

## 4 Design

Using the approach described in the previous section requires us to answer three questions:

- In response to a request from a client, how can a web server accurately identify the list of dependent resources that it should inform the client about, including ones hosted in other domains, without having clients fetch resources that are irrelevant to the ongoing page load (thereby wasting bandwidth)?
- How can a server provide a sufficient number of dependency hints to clients, without knowledge of how content is personalized by other domains that serve resources for a given page?
- At any point in time during a page load, a client's access link bandwidth is shared by resources that are explicitly fetched by the client or proactively pushed by servers. Given this contention, when should clients schedule resource fetches? What resources should servers push?

In designing VROOM to address these questions, we respect two primary constraints: 1) we do not rely on input from developers to characterize the dependencies on web pages because the resources on a page are typically spread across several domains [21], and no single developer is likely to have complete knowledge about all dependencies on a page; and 2) to preserve the integrity of content and to protect user privacy, any client will accept a resource only from the domain that serves that resource, and it will share its cookie for a domain only with web servers in that domain. Figure 6 illustrates the server-side and client-side components of VROOM, which we describe next.

### 4.1 Server-side dependency resolution

Generating an accurate list of dependent resources for a web page is challenging due to the constant flux in resources on modern pages. Moreover, unlike recent work [22, 35] focused on generating a page's stable dependency *structure*, here we need to identify the precise *URLs* of resources that a server must either push or include in its dependency hints. To appreciate the challenge in doing so, we first consider two strawman approaches before describing our solution.

#### 4.1.1 Strawmans for resource discovery

**Strawman 1 (Online).** When a server receives a request for the HTML of a page, the server could load the entire page locally, mimicking a client browser, to identify the other resources on the page. However, many of the URLs fetched by the server during its page load will not be requested during the client's page load. On

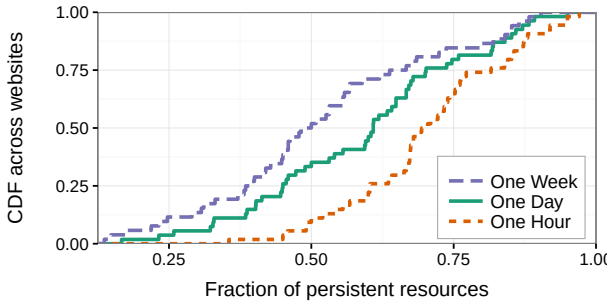



**Figure 7: Fraction of resources, per page in the Alexa Top 100, that persist over different time scales.**

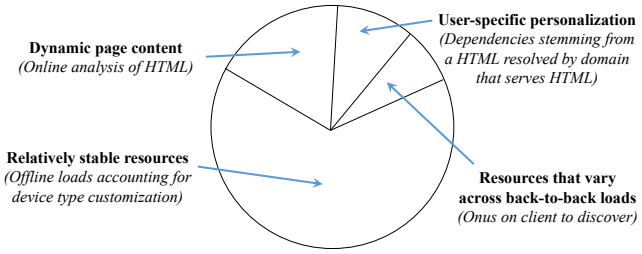



**Figure 8: Summary of techniques used in VROOM for server-side dependency resolution to account for the different types of resource on any web page.**

the one hand, back-to-back loads of a page often differ in the exact set of URLs fetched (e.g., ads typically insert a randomly selected identifier into the URLs they fetch). For example, 22% of the URLs fetched to load the median page in the Alexa Top 100 list change across back-to-back loads. On the other hand, loading a page at one server cannot account for the personalization performed by other domains, since the server only has the user's cookie for its domain. If servers fail to account for these discrepancies and either push to the client or ask the client to fetch unnecessary objects, the user is likely to experience *higher* load times.

**Strawman 2 (Offline).** Alternatively, a server can periodically load each page that it serves. This enables the server to account for the variation in resources over time; when a client requests the HTML for a page, the server can return the set of resources that it has repeatedly observed on recent loads of that page. With this approach, we risk missing a large fraction of URLs that a client will need to fetch when loading the page. For example, the set of stories or set of products on the landing page of a News or Shopping site changes often. Figure 7 confirms this; for the median site in the Alexa Top 100, only 70% of the resources on the landing page remain stable over one hour, and this number drops to 50% over one week.

### 4.1.2 Our solution: offline + online discovery

The two strawmen approaches for server-side resource discovery illustrate the following trade-off: servers must ensure that a client does not end up fetching unnecessary resources, but if they are too conservative (and thus let clients discover many resources on their own), the utility of input from servers will be minimal. To address this trade-off, we observe that *both* offline and online dependency resolution are necessary at the server. Figure 8 summarizes the techniques we employ. Periodic offline resolution of a page helps

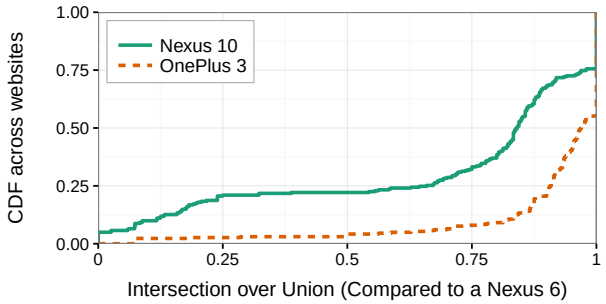



**Figure 9: Comparison of the stable set of resources on each page when the user device is a Nexus 6 compared with when the user device is a Nexus 10 or a OnePlus 3.**

confirm which URLs are consistently fetched when loading the page, whereas online resolution helps account for flux in page content.

**Offline dependency resolution.** Offline server-side determination of the resources on a page works as described previously: the page is loaded periodically (once every hour in our implementation), and at any point in time, URLs fetched in all recent loads are considered likely to also be fetched when a client loads the page. However, even the largely stable subset of resources on a page can vary across different types of client devices (Figure 9). For example, it is common for website providers to use CSS stylesheets and JavaScript objects that cause different clients to fetch images of different sizes on the same page depending on the client's display resolution or pixel density.

VROOM's offline server-side dependency resolution efficiently accounts for device-specific customization of resources in two ways. First, the server need not load each page on every type of device; this would be onerous given the large variety of smartphones and tablets on the market. Instead, after a few loads of a page, the server can bin all device types into a few equivalence classes. The equivalence classes can vary across pages because different pages may be customized based on different device characteristics. For example, in Figure 9, the stable set of URLs fetched when loading a page on a Nexus 6 smartphone matches the stable set of resources for a OnePlus 3 phone much more closely than for a Nexus 10 tablet. Second, after device type equivalence classes for a page are identified, the server need not load the page on a real device in each class. Instead, the server can leverage existing device emulation tools [3].

**Online HTML analysis.** In addition to offline dependency resolution, when a VROOM-compliant web server responds to a request with an HTML object, it not only informs the client of dependencies discovered from loading this object offline, but also includes all URLs seen in the HTML object by parsing it on the fly. While there can be other sources of dynamism on a page (e.g., a script on the landing page of a shopping site may fetch products currently on sale), we show later in Section 6 that accounting for the URLs in HTML objects suffices on most pages to capture the flux in page content. Importantly, we find that server-side parsing of HTML objects as they are being served adds a median delay of only roughly 100 ms across the landing pages of the top 1000 websites. This overhead is offset by the multi-second reduction in page load times made possible by server-aided resource discovery.

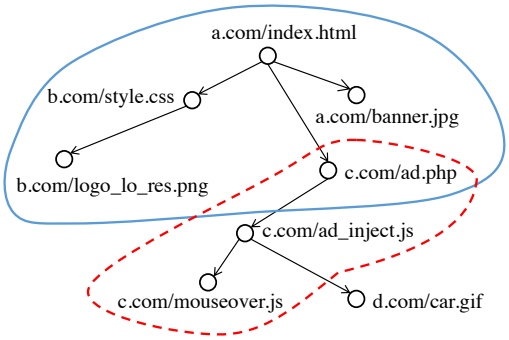



**Figure 10: Illustration of how VROOM-compliant servers account for personalization. A client that requests for `index.html` from `a.com` only discovers the resources within the solid blue envelope, because a request for `ad.php` returns a HTML, whose content could be personalized to a specific user. The client discovers the resources in the dashed red envelope in response to its request for `ad.php` sent to `c.com`. The client will have to itself discover the need to fetch `d.com/cars.gif`.**

### 4.2 Accounting for personalization

Unlike content push, dependency hints allow a server to inform clients about resources served by other domains. But, content served by one domain may be personalized in ways that other domains are unaware of (e.g., based on information stored in cookies).

A naive solution for handling personalization would be for any web server to never return dependencies derived from external content. In other words, any resource that is discovered during the page load by parsing or executing an external resource is deemed as one that could differ due to personalization. For example, in Figure 10, in response to a request for `index.html`, `a.com` can inform a client about `b.com/style.css`, but let the client discover the need to fetch `b.com/logo_lo_res.png` only when it requests `style.css` from `b.com`. Accounting for personalization in this manner would however inflate the latency incurred by the client in discovering all resources on the page, thereby limiting its ability to fully utilize the network.

To handle content personalization while enabling low-latency resource discovery, we observe that websites are personalized primarily in two ways: by customizing the content of HTML responses,[3] and by adapting the execution of scripts. Server-side resource discovery in VROOM accounts for these two types of personalization as follows. First, web servers omit hints for dependencies derived from an external resource only when that resource is an HTML object (i.e., an embedded iframe); servers do include dependencies derived from other types of external resources (e.g., `style.css` in Figure 10). In comparison with the naive solution described above, our approach reduces the latency for the client to discover resources. Second, of the resources determined based on JavaScript execution, those affected by user-specific state (e.g., local time) are left to clients to discover on their own. JavaScript-based personalization will typically vary over time, and hence, such unstable resources will get filtered out via offline dependency resolution.

[3]Content of CSS stylesheets and scripts are seldom user-specific. Personalization of images and videos (i.e., returning different versions when the same URL is fetched) is typically device-specific, which we have previously accounted for.

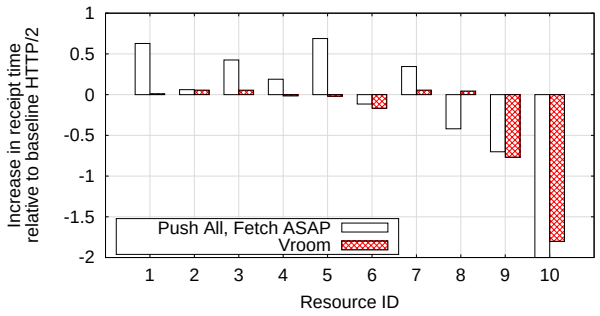



**Figure 11: Need for and utility of careful scheduling of server-side push and client-side fetch of the resources that need to be processed (HTML, CSS, and JS) on http://eurosport.com. Resources are ordered based on the order in which they are fetched with baseline HTTP/2.**

### 4.3 Cooperative request scheduling

Finally, we turn our attention to questions that must be answered for clients to benefit from server-side resource discovery. How should web servers combine the use of HTTP/2 PUSH and dependency hints to aid clients? How should clients utilize the dependency hints from servers?

**Strawman: Push whenever possible. Fetch upon discovery.** We first consider the most straightforward answers to these questions. When a web server receives a client's request for an HTML object, it can push to the client all dependent resources that it owns. The server can inform the client of all other dependencies via dependency hints. As soon as the client receives these hints, it can initiate downloads for all specified URLs.

**Problem: Bandwidth contention.** Though this simple strategy significantly improves utilization of the client's access link, we do not see a commensurate increase in CPU utilization. This is because, though the client receives the complete set of resources on a page well before it would without server push and dependency hints, contention on the client's access link slows down fetches of some of the resources that need to be processed. In the example in Figure 11, though the time to fetch the first 10 resources that need to be processed reduces by 2 seconds with the "Push All, Fetch ASAP" strawman, simultaneous use of the client's access link to transfer all of these resources delays the first few resources, causing the browser's processing to stall. Due to the resulting under-utilization of the CPU, we show later in Section 6 that applying this strawman solution yields no improvements in page load times.

**Solution: Prioritization via selective push and staged downloads.** Our solution to this problem is to prioritize the fetches of resources that need to be processed (HTML, CSS, and JS) over those that need not be processed (e.g., images and videos).[4] Classifying resources into high and low priority groups helps because, once the client has finished fetching all resources that need to be processed, utilization of the CPU and the network are largely decoupled over the rest of the page load. Moreover, the types of resources that need to be processed typically constitute a small fraction of the bytes on a page [7].

[4]We however consider all resources that are descendants of third-party HTML objects (i.e., HTMLs within a page's iframes) as low priority because web browsers process iframes only after the root HTML for the page has been completely downloaded and parsed. This helps minimize network contention for high priority resources referenced in the root HTML, thus reducing the fetch times for those resources.

| Header | Description |
|---|---|
| Link preload | Resources to be processed (e.g., JavaScript and HTML objects), fetched at the highest priority |
| x-semi-important | Resources to be processed that are lazily fetched, e.g., "async" JavaScript or CSS objects |
| x-unimportant | Resources that do not need to be parsed or executed (cannot have derived children), e.g., images |

**Table 1: HTTP headers used by VROOM-compliant servers to provide dependency hints to client browsers. Headers are listed in decreasing order of priority. Resources in each header are listed in the order they need to be processed.**

This prioritization of resources that need to be processed is achieved in VROOM via two means. First, when a web server responds to a client's request for an HTML object, out of all the dependencies the server identifies, it pushes the content of only the high priority resources served from the local domain. All other dependencies are returned to the client via dependency hints. Second, when the client receives a list of URLs, it immediately fetches only high priority resources; the server's hints specify resources in the order in which the client will need to process them, so client requests simply mimic that order. Once resource discovery from servers is complete and the client has finished fetching all high priority resources that have been discovered, it issues requests for all other resources at once. In Figure 11, the time by when receipt of the 10 resources shown completes is the same with VROOM's scheduling as with the strawman strategy, but without significantly delaying the receipt of any individual resource.

Note that VROOM's scheduler is tailored for the setting where web pages are loaded on a state-of-the-art mobile device connected to a LTE network; as we saw earlier in Section 2, the client CPU is the bottleneck in this case. Alternate scheduling strategies will likely be necessary in settings where either network bandwidth (e.g., because of many users simultaneously accessing the cellular network [23]) or latency (e.g., on 2G or 3G networks [25]) is the bottleneck.

## 5 Implementation

The realization of VROOM requires both server-side (offline and online dependency resolution; pushing resources to clients and including dependency hints in responses) and client-side (scheduling fetches of hinted objects) changes. Since this preempts evaluation in the wild, we have implemented VROOM to accelerate web page loads when Google Chrome is used to load pages from the Mahimahi [36] replay environment.

### 5.1 Server-aided resource discovery

VROOM-compliant servers inform clients of dependent resources via content pushes and dependency hints. To push resources, we leverage HTTP/2's PUSH capability; since Mahimahi uses Apache web servers which do not yet support HTTP/2, we run an HTTP/2 nghttpx reverse proxy [14] in front of each web server. For dependency hints, we rely on embedding additional headers in HTTP responses. For example, when a browser encounters a Link preload header [16] in an HTTP response, the browser will immediately issue a request for the URL embedded in that header.[5] Table 1 summarizes the headers used by VROOM to provide dependency hints to clients.

To minimize stalls in the client browser's processing of resources, dependency hints from any server list resources in the order the client will need to process them and the client requests hinted resources in this order; the server discovers this order during its offline and online dependency resolution. However, since the client issues these requests back-to-back, a web server may receive multiple requests near-simultaneously, causing it to respond to all requests in parallel. The resulting contention for the client's access link bandwidth leads to the slowdown of some resources over others as seen with the "Push All, Fetch ASAP" strategy in Figure 11. Therefore, we modify Mahimahi so that any web server returns the content for requested resources in the same order in which it receives requests.

### 5.2 Scheduling requests with JavaScript

To schedule client-side downloads of URLs learned via dependency hints (Section 4.3), VROOM uses a JavaScript-based request scheduler. For each recorded page in Mahimahi, we modify the page's top-level HTML using Beautiful Soup [2]. VROOM's scheduler script is added as the first tag in the HTML, thereby ensuring that the browser executes this script as soon as it begins parsing the HTML.

VROOM's scheduler script begins its execution with two steps. First, it defines an `onload` handler, *response_handler*, which it attaches to each request that it makes. This handler maintains a list of dependency hints that it has seen and fires every time the browser receives a response for a request made by VROOM's scheduler. Second, the script issues an XHR (*XMLHttpRequest*) for the page's HTML, whose URL we embed as an attribute in the `<html>` tag.[6]

The scheduler examines Link preload headers in the response for the page's HTML to discover dependency hints.[7] It then issues requests for high priority dependencies by adding `<link>` tags (with the preload attribute set) to the DOM. Importantly, the scheduler's requests for high priority resources are served from the browser's local cache, because the browser itself immediately requests URLs included in Link preload headers. Moreover, modern browsers permit only a single outstanding request for any given URL.

Thereafter, VROOM's scheduler runs in an event-driven loop. Whenever the browser invokes *response_handler* upon receiving a resource, the scheduler marks that resource as fetched. The scheduler then examines the set of outstanding requests to determine whether it should start fetching dependencies in the next level of priority. Specifically, once all high priority resources learned via dependency hints have been received, VROOM's scheduler issues requests for all semi-important resources that it has discovered until that point. This process then repeats for low priority resources.

By using a JavaScript-based request scheduler, our implementation of VROOM can accelerate page loads on unmodified commodity browsers. However, due to the single-threaded nature of Javascript, if

[5] While Link preload headers are currently supported by Chrome, other browsers such as Firefox are actively in the process of adding support for these headers: https://bugzilla.mozilla.org/show_bug.cgi?id=1222633.

[6] Note that VROOM's scheduler removes this attribute and its own DOM node from the page once the XHR for the top-level HTML is issued. Thus, subsequent accesses to the DOM are not affected.

[7] To ensure that the request scheduler can securely access headers in HTTP responses served from third-party domains, responses must include the "Access-Control-Expose-Headers" header with the values "Link," and our custom headers "x-semi-important" and "x-unimportant."

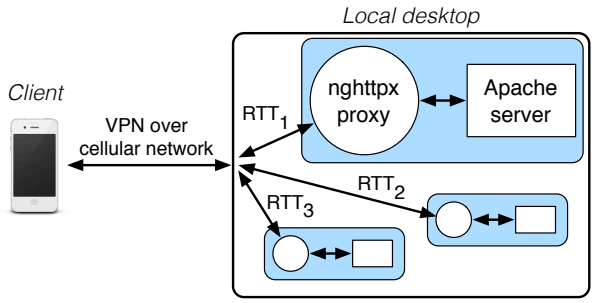



**Figure 12: Setup to evaluate page load performance enabled by our implementation of VROOM.**

the browser is executing another script on the page when a response arrives, *response_handler* will not fire immediately. This delays the fetches of lower priority resources. Therefore, in the future, incorporation of the scheduling logic into the browser may enable greater performance gains than what we report.

## 6 Evaluation

We evaluate VROOM from two perspectives: 1) performance benefits for users, and 2) the accuracy with which servers can aid resource discovery for clients. The key highlights from our evaluation are:

- Across 100 popular News and Sports websites, we see that the adoption of VROOM would yield near-optimal performance on the median site with respect to two different metrics: page load time (PLT) and above-the-fold time (AFT). In comparison to the adoption of only HTTP/2, VROOM's use would reduce the median PLT and AFT values by 30% and 20%, respectively.
- Simple alternatives to VROOM (e.g., relying only on prior loads of the page for dependency discovery, using only HTTP/2 PUSH but not dependency hints, or not scheduling pushes and fetches) can increase the median page load time by over 2 seconds.
- On the median site, VROOM's server-side discovery of dependent resources has a false negative rate below 5%, which in turn results in a 22% decrease in client-side latency to discover all resources on the page.

### 6.1 Impact on client performance

**Methodology.** We use the setup shown in Figure 12 to experimentally evaluate our implementation of VROOM. We load pages in Chrome for Android on a Nexus 6 smartphone connected to a Verizon LTE hotspot. The phone is also connected via USB to a desktop, which subscribes to events exported by Chrome via the Remote Debugging Protocol (RDP). The phone has a VPN tunnel setup to the desktop, on which we host Mahimahi [36]. For every web page on which we test VROOM, we initially load the page with the desktop as a proxy to have Mahimahi record all page contents; page load times with this setup match those measured when we load pages with the phone directly communicating with web servers. Thereafter, when replaying page loads, we configure Mahimahi such that traffic between the phone and any of the web servers is subjected to not only the delay over the cellular network but also the median RTT observed between the desktop and the corresponding web server when recording page contents. The desktop on which we deploy Mahimahi is sufficiently well-provisioned so that its CPU or network is not a bottleneck, and as we mentioned earlier in Section 2, load times when we replay page loads using HTTP/1.1 between the client and all servers closely match load times measured when loading pages directly from the web.

We evaluate the utility of VROOM on the landing pages of the top 50 News and top 50 Sports websites as well as 100 randomly chosen sites from Alexa's top 400 sites;[8] in most of our experiments, we focus on the News and Sports sites because, as seen earlier in Section 2, the need for performance improvements on these sites is particularly acute with a median page load time of 10.5 seconds. We load each page 3 times in our replay setup and consider the load with the median page load time. During these loads, web servers identify the dependencies to return to clients by drawing upon three prior loads of each page gathered 1, 2, and 3 hours prior to when we recorded the page content in Mahimahi.

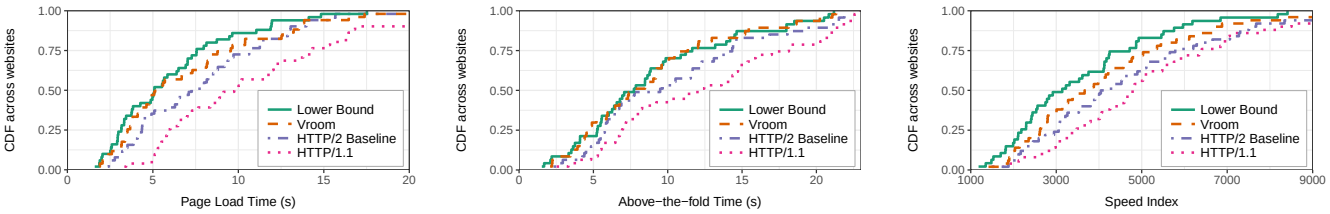
In addition to page load times, we also evaluate VROOM's benefits with respect to additional metrics which capture the speed with which page content is rendered, as this impacts users' perception of page load performance [29]. To measure these metrics, we use *screenrecord*, a utility program which captures videos of page loads on Android devices. We pass the recorded videos to the *visualmetrics* tool [18] which outputs the metrics of interest.

**Improvement in page load times.** First, we compare page load performance when using VROOM with that when doing a HTTP/2 based replay (i.e., same setup as Figure 12, except that servers simply return requested resources), which we refer to as the HTTP/2 baseline. On the top 50 News and top 50 Sports sites, Figure 13(a) shows that VROOM significantly reduces the page load time on the median site—from 7.3s with the HTTP/2 baseline to 5.1s with VROOM—closely matching the lower bound (median of 5s); the lower bound corresponds to the maximum of the CPU-bound and network-bound loads described earlier in Section 2. On the 100 sites from the top 400, where the median page load time is significantly lower than on the News and Sports sites even with the HTTP/2 baseline (median of 4.8s), VROOM still reduces the median page load time to 4s.

The improvements in load times described above are feasible when all domains adopt the changes prescribed by VROOM. To understand VROOM's benefits as it is incrementally adopted, we evaluate VROOM in a scenario where the first party domain for every web page (i.e., the domain that serves the root HTML for the page), along with all other domains controlled by the same organization, are VROOM-compliant. All other domains contacted in each page load only conform to HTTP/2, without pushing content or providing dependency hints. While the median page load time across the News and Sports increases to 5.6s in this case, as compared to 5.1s with universal adoption of VROOM, this is still significantly lower than the 7.3s median page load time with the HTTP/2 baseline.

We also compared VROOM to Polaris [35], a state-of-the-art web accelerator which uses information about a page's dependency structure to prioritize requests for critical resources. Figure 14 shows that, compared to Polaris, VROOM is able to reduce the median page load time for the popular News and Sports sites from 6.4s to 5.1s. As noted in Section 2, the primary reason for this performance gain

[8] The high page load times for these pages and the need to load each page multiple times to account for the variability of the cellular network limit us from running our evaluation on more web pages.




(a) (b) (c)

**Figure 13: With respect to three different metrics, VROOM yields significant benefits compared to simply upgrading from HTTP/1.1 to HTTP/2, and comes close to matching the achievable lower bound.**

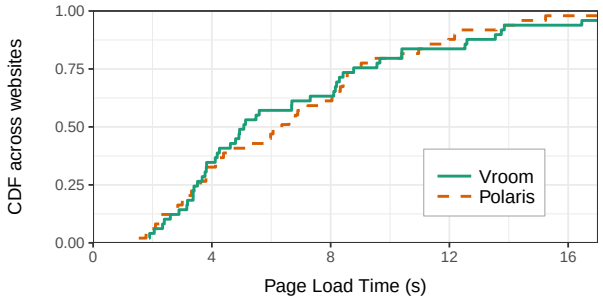


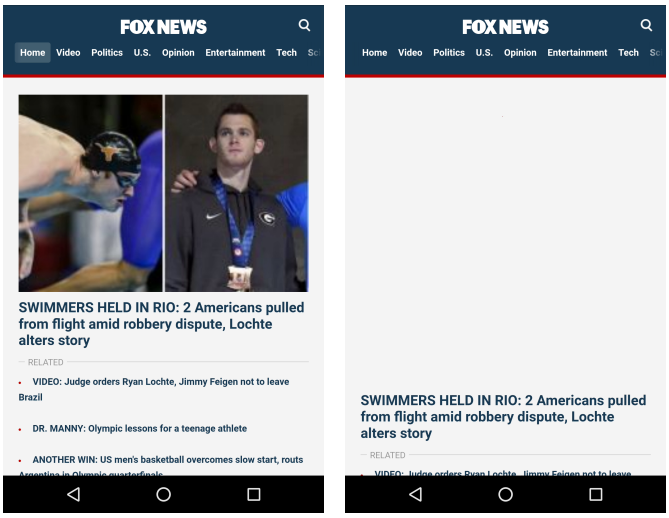

**Figure 14: Comparison of page load times when pages are loaded with VROOM and with Polaris.**

is that Polaris still leaves clients to discover resources on their own (i.e., the client can discover the need to fetch a resource only after fetching and evaluating another resource). Further, Polaris does not specify policies for servers to proactively push resources to clients in anticipation of future requests.

Though Figures 13(a) and 14 show that VROOM significantly improves performance for the median page, these benefits are marginal at the tail; in fact, Polaris outperforms VROOM in the tail of the load time distribution. The reasons for this are two-fold. First, certain sites include dynamic content that VROOM is unable to detect simply by online analysis of HTMLs; VROOM defers the discovery of such unpredictable resources to clients and is unable to provide hints for these resources. Second, when clients prefetch objects (specified by dependency hints) and servers in multiple domains concurrently push resources, bandwidth contention can result in high priority resources being delayed. These results illustrate that combining the complementary approaches used in VROOM and Polaris is a promising direction of future work.

**Improvement in visual performance metrics.** In addition to reducing page load times, VROOM also improves metrics such as Above-the-fold time and Speed Index which grade performance based on visual completeness of page loads. Above-the-fold time measures the time until all content that is "above the fold" (i.e., a user sees prior to scrolling) is rendered to its final state. Speed Index extends this metric to capture the rate at which all the content that is above the fold gets rendered. Figures 13(b) and (c) show that, across the popular News and Sports sites, VROOM improves the Above-the-fold time and Speed Index for the median site by 400ms and 380ms, respectively.

Figures 15 shows these benefits on an example site (http://m.foxnews.com/). With VROOM, rendering of all above the fold content completes at 9.26s. At that same time in the page load with the




(a) (b)

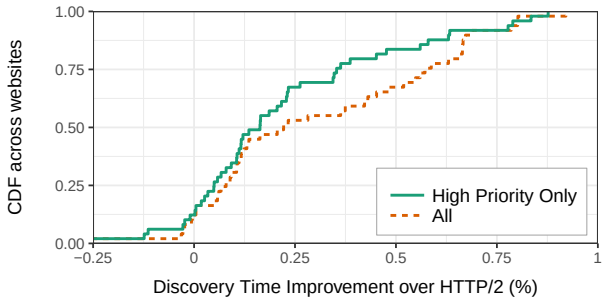
**Figure 15: For the Fox News mobile site, (a) rendering of the above the fold content completes at 9.26s with VROOM; (b) with only HTTP/2 enabled, rendering is incomplete at that time and completes only later at 13.87s.**

HTTP/2 baseline configuration, the main images are still missing and the rendering of the page is yet to converge. With HTTP/2, rendering of all above the fold content takes 4.6s longer, completing at 13.87s into the page load.

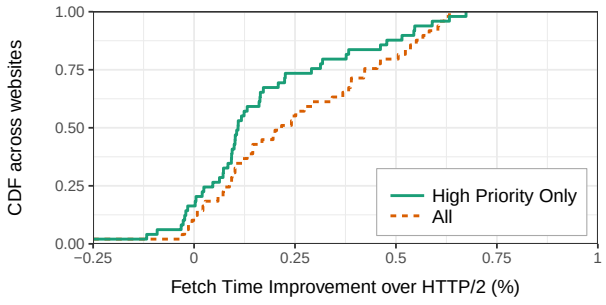
The above-described improvements in page load performance with VROOM are due to several reasons; we dig into each of these next. For brevity, we show results only for the popular News and Sports sites.

**Latency in discovering and fetching resources.** A key benefit of server-aided resource discovery is that it enables clients to discover resources and complete fetching them much sooner than in normal page loads. Figure 16 depicts these improvements both when considering all resources identified as dependencies by VROOM-compliant web servers, and also when considering only those dependencies which are high priority resources (i.e., HTML, CSS, and JS objects, which are the ones that need to be parsed or executed).

Reducing the time by when the client completes fetching all dependencies (a 22% reduction on the median page) is crucial because any further network activity necessary to complete the page load is only to fetch the small subset of unpredictable resources that servers fail to identify as dependencies. But, in addition, the drop in the time by when all high priority dependencies finish downloading—a 12%




(a)




(b)

**Figure 16: In comparison to HTTP/2, VROOM reduces the latency in both (a) discovering resources and (b) completing their downloads.**

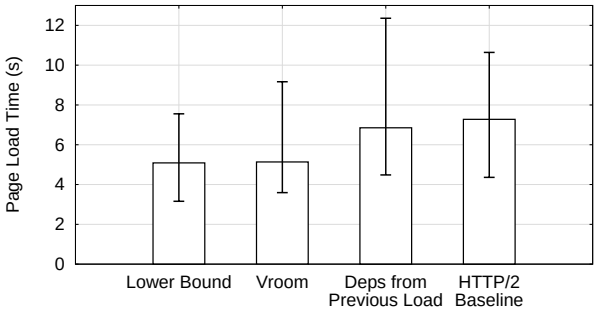



**Figure 17: In contrast to VROOM, if servers return dependencies as all the resources seen on a prior load of the page, page load times increase on many pages. $25^{th}$ percentile, median, and $75^{th}$ percentile are shown in each case.**

median reduction compared to baseline HTTP/2—is also critical since use of the CPU and the network are largely decoupled thereafter. Both of these improvements are made possible because of the speedup in the client discovering dependencies: in the median, 22% and 16% faster discovery of all dependencies and of all high priority dependencies, respectively.

Faster discovery and preemptive receipt of resources also helps reduce the time spent on the critical path waiting to receive data over the network. While the simple use of HTTP/2 led to network delays accounting for over 30% of the critical path on the median site (Figure 4), VROOM's use reduces the network wait time on the critical path by 24% on the median site.

**Utility of accurate dependency inference.** While input from servers enables clients to discover and fetch dependent resources sooner, the benefit of this input strongly relies upon the accuracy of the dependencies returned. To show this, we consider servers identifying

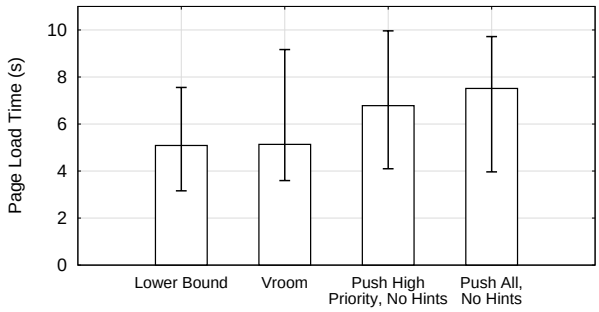



**Figure 18: VROOM improves page load times compared to using only server-side push to inform clients of dependent resources. $25^{th}$ percentile, median, and $75^{th}$ percentile are shown in each case.**

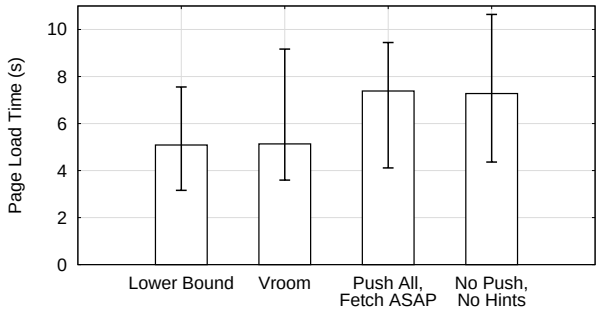



**Figure 19: VROOM's judicious scheduling of server push and client fetch is key to enabling improvements in page load time. $25^{th}$ percentile, median, and $75^{th}$ percentile are shown in each case.**

the set of dependencies to return to a client simply based on a prior load of the page, i.e., all resources seen in a load within the past hour are assumed to be relevant to the new load. Figure 17 shows that, though the median page load time does reduce with this approach, the extraneous inaccurate dependencies returned to the client degrade performance on many sites; the $75^{th}$ percentile increases by over 1.5 seconds.

**Need for combining HTTP/2 PUSH and dependency hints.** Beyond accuracy in server-side dependency discovery, VROOM's benefits draw upon the *combined* use of HTTP/2 PUSH and dependency hints. Figure 18 shows that simply relying on server push is insufficient. Irrespective of whether we push all static resources or only the subset of static resources that need to be processed, median page load time remains more than 2 seconds higher than with VROOM. This stems from the preponderance of third-party resources on modern web pages; servers can inform clients of such dependencies only via dependency hints, as any server can securely push only the content that it hosts.

**Utility of scheduling.** While HTTP/2 PUSH and dependency hints enable faster resource discovery, performance improvements with VROOM also hinge upon judicious coordinated scheduling of pushes and downloads of discovered dependencies. Figure 19 illustrates the utility of the cooperative scheduling in VROOM compared to the strawman "Push All, Fetch ASAP" approach discussed in Section 4.3 (where servers push any resource they can and clients fetch any resource immediately upon discovery). Because only high priority resources are pushed by VROOM servers and are preferentially fetched by clients, contention for access link bandwidth has minimal impact on the processing of resources. The resultant increased

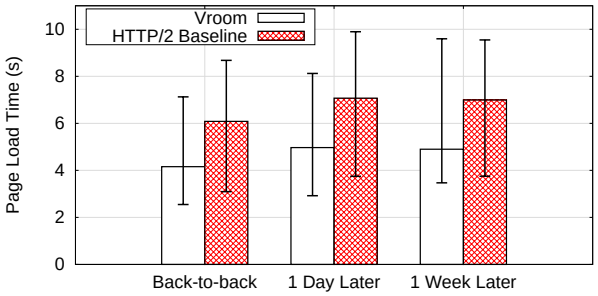



**Figure 20: For three different delays between the load that warms the browser's cache and the load on which we evaluate page load performance, VROOM reduces page load times. $25^{th}$ percentile, median, and $75^{th}$ percentile are shown in each case.**

utilization of the CPU enables VROOM to improve performance over baseline HTTP/2. Whereas, use of the strawman approach for servers to inform clients of dependencies offers minimal benefits; in fact, the median load time increases as a result of increased network contention.

**VROOM accelerates page loads with warm caches.** All of our experiments thus far have considered the browser's cache to be empty. To evaluate VROOM when the browser's cache is not empty, we first identify cacheable objects by examining the headers in HTTP responses. We mimic three different scenarios, wherein once a page is loaded by a client, it loads the page again immediately thereafter (i.e., back-to-back loads), a day later, or a week later. Importantly, to prevent wasted bandwidth, resources that were already cached at the client were not pushed by servers.

Figure 20 shows that VROOM significantly improves page load times in all three warm browser cache settings. When considering back-to-back loads, VROOM reduces the page load time of the median site by 1.6s. This improvement increases to 2.2s and 2.1s for the loads separated by one day and one week, respectively.

### 6.2 Accuracy of server-side dependency resolution

**Setup.** To evaluate the accuracy of the resource dependencies that VROOM-compliant servers return to clients, we consider 265 web pages drawn from popular News and Sports websites; these pages span a variety of page types such as landing pages, individual articles, results for specific games, etc. We load these pages once every hour for a week from the perspective of four users, whose cookies are seeded by visiting the landing pages of the top 50 pages in the Business, Health, Computers, and Shopping/Vehicles Alexa categories, respectively. Every hour, we load each page twice back-to-back from every user's perspective for reasons described shortly.

As described earlier in Section 4.1, server-side dependency resolution in VROOM relies upon both offline and online analysis. The dependencies identified via offline dependency resolution include the resources seen in each of the loads in the past 3 hours. For online analysis, we model the server which serves any HTML object—either the root HTML on a page or one embedded in an iframe—returning all links in that HTML. Recall that, in order to account for personalization, VROOM-compliant servers return dependencies—either via push or via dependency hints—only in response to requests for HTML objects (Section 4.2).

**Strategies for server-side resource discovery.** We compare dependency resolution in VROOM with both the strawman approaches described earlier in Section 4.1: *offline-only* returns URLs seen in the intersection of loads over the past 3 hours, and *online-only* loads the page on the fly at the server and returns the URLs fetched.

**Definition of accuracy.** To evaluate the accuracy with which each of these approaches can identify the set of URLs that a client must fetch during a page load, we partition the set of URLs we see in any page load into a predictable and unpredictable subset. We identify the subset of unpredictable URLs as ones that differ between back-to-back loads; these are URLs that VROOM leaves it up to the client to discover. As seen in Figure 21(a), out of the subset of resources on a page that a server can potentially return as dependencies in response to a request for a HTML (i.e., all the resources derived from HTML minus the ones derived from embedded iframes), the predictable subset accounts for over 80% and over 95%, respectively, in terms of the number of resources and the number of bytes. We evaluate the accuracy of each approach for server-side resource discovery with two metrics—the number of resources identified as dependencies by the server which do not appear in the predictable subset of the client's load (false positives), and the number of resources in the predictable subset that the server fails to identify (false negatives)—both computed as fractions of the predictable subset's size.

**Results.** First, Figure 21(b) shows that, out of the resources in the predictable subset, the fraction that VROOM-compliant servers would fail to identify is less than 5% for the median page. Whereas, offline-only dependency resolution ends up missing as many as 40% of the predictable subset of resources seen on any particular page load because of its inability to cope with changes from hour to hour. The online-only approach is perfect with respect to this metric, which validates our design decision to account for personalization by limiting the set of dependencies returned to exclude resources recursively derived from embedded HTMLs.

Second, with respect to the overhead imposed on clients by returning dependencies not in the predictable subset, Figure 21(c) shows that VROOM matches the offline-only approach. In both of these cases, identifying the stable set of resources seen consistently on a page helps in ignoring resources that happen to show up on a single load. Due to its inability to cope with such nondeterminism, the online-only approach identifies many extra resources, which inflate the predictable subset by as much as 20% in the median case.

## 7 Discussion

**Deployability.** Unlike attempts at clean-slate redesigns of the Internet's architecture, new designs for client-server interactions on the web are more amenable to deployment, as evidenced by the recent incorporation of SPDY into HTTP/2. This is possible because of several differences between the web and general communication over the Internet: 1) clients directly interact with web servers, unlike an ISP having to rely on other ISPs to forward traffic, 2) a few popular browsers and web servers are dominant, and 3) since some browsers (Chrome and IE) are controlled by popular content providers (Google and Bing), these providers can unilaterally test performance improvements enabled by a new proposal such as ours without depending on adoption by others.

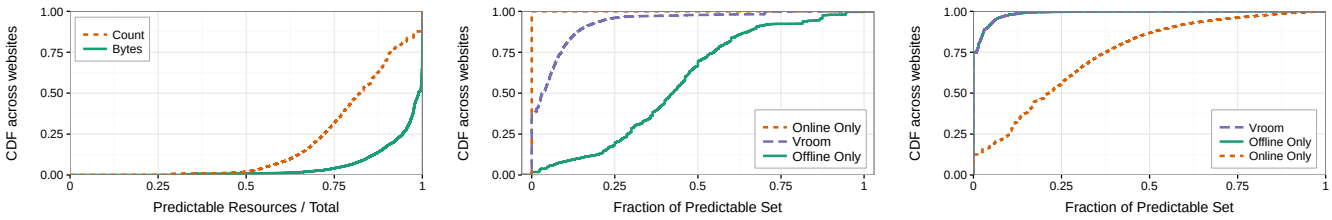



**Figure 21: (a) Among the resources derived from a page's root HTML, except for those derived from embedded HTMLs, the contribution of the predictable subset to the number of resources and bytes. As a fraction of the size of this predictable subset, the resources that are either (b) missed or (c) are extraneous when using VROOM's server-side dependency resolution as compared to offline-only and online-only analyses.**

**Server-side overhead.** VROOM-enabled servers identify dependent resources using both online and offline analyses. For popular websites that host thousands of web pages, loading each page every hour to facilitate offline dependency resolution will likely be onerous. We observe that there are typically only a few *types* of pages on each site and the stable set of resources (e.g., CSS stylesheets, fonts, logo images, etc.) are likely to be common across pages of the same type. For example, on a news site, landing pages for different news categories are likely to share similarities as will news articles about different individual stories. We defer for future work the task of leveraging the similarity across pages of the same type to improve the scalability of VROOM's server-side resource discovery.

Note that the overhead of resolving dependencies will be incurred only by the domains which serve HTML objects, i.e., the root HTML for an entire page or for an individual frame in the page. Other servers involved in a page load need only serve individual resources as they are requested. Moreover, for over 80% of sites, the landing page's HTML is served directly from origin web servers [7] and not from third-party CDNs. Thus, the overhead imposed by VROOM's resource discovery will largely be incurred by top-level domains who are only responsible for the websites they own.

**Security.** At first glance, it may appear that having servers push resources and having clients fetch hinted resources present new security concerns for page loads, whereby compromised web servers could push or provide hints for malware. However, of the resources that are pushed or fetched based on dependency hints, client browsers will only process those resources referenced by the page being loaded, e.g., as HTML tags. Further, page loads that include unnecessary pushes or hints will still load correctly to completion, albeit with higher load times due to the downloads of unnecessary resources. Thus, page loads with VROOM encounter the same (but not worse) security concerns as page loads do today.

## 8 Related work

**Mobile web performance.** Prior measurement studies [34, 44] have analyzed the performance of mobile web browsers. Like us, these studies find that CPU *and* network delays are bottlenecks when loading pages on mobile devices and high-latency cellular links.

Some new proposals aim to alleviate the effects of these bottlenecks by altering how pages are written and served. For example, Google's AMP project [1] asynchronously fetches many resources required to load a page, and uses Google's CDN to serve AMP-enabled content with HTTP/2. In contrast to AMP, VROOM can speed up the loads of legacy web pages. VROOM can also improve the performance of AMP-based pages by enabling asynchronous fetches earlier using server-provided hints.

**Dependencies in page loads.** Many recent systems [22, 30, 35] use offline analysis to discover dependencies inherent to web pages. Due to the dynamic nature of web content, previously generated dependency graphs can only capture the structure of a page, but not the exact set of URLs that must be fetched in any particular load of the page. These systems that leverage a priori knowledge of inter-object dependencies can therefore only reorder requests for resources, but leave the onus of discovering resources on the client. VROOM overcomes this limitation by combining offline dependency resolution with online analysis and by carefully spreading resource discovery across domains.

WProf [41] identifies dependencies between different browser components (e.g., HTML parser, JavaScript engine) that arise during page loads and degrade performance. By leveraging HTTP/2 PUSH and browser support for fetching dependency hints, VROOM reduces the coupling between the CPU and the network; processing a resource is rarely blocked by fetching, and vice versa.

**Proxy-based acceleration.** Cloud browsers improve mobile web performance by dividing the load process between the client's device and a remote proxy server. By resolving dependencies using a proxy's wired connections to origin servers (in place of the client's slow access link), such systems can significantly reduce page load times [15, 22, 36, 40, 43]. However, the reliance on web proxies raises security and privacy concerns, as clients must share their cookies with the proxy in order to preserve personalization and must trust that the proxy preserves the integrity of content served over HTTPS.

**Network optimizations for faster page loads.** HTTP/2 [20] (formerly SPDY [17]) reduces load times by allowing client browsers to multiplex all requests to an origin on a single TCP connection. HTTP/2 also allows servers to speculatively "push" objects they own before the user requests them (saving RTTs) [39]. VROOM demonstrates the need for HTTP/2's PUSH feature to be combined with dependency hints in order to securely speed up dependency resolution on the client.

Recent work has isolated two distinct factors that limit performance improvements with HTTP/2: dependencies inherent in web pages and browsers restrict HTTP/2's ability to reduce load times [42], and the use of a single TCP connection can be detrimental in the presence of high packet loss [24]. VROOM would benefit from multiplexing requests on the same connection, but it can be used with HTTP/1.1 in the face of high packet loss.

**Client-side optimizations.** Content prefetching and speculative loading systems reduce the effect that high network latencies have on web performance [26, 28, 37, 45]. These systems predict user browsing behavior and speculatively fetch content in hopes that users will soon do the same. However, accurately predicting user browsing behavior remains a challenge. Thus, prefetching often leads to wasted device energy and data usage [38].

Other client-side improvements reduce energy usage and computational delays using parallel web browsers [31, 32] and improved hardware [47]. By increasing the amount of parallelization for necessary page load tasks (e.g., rendering), these systems reduce energy usage and have positive impacts on page load times.

Each of these optimizations is complementary to our work. However, VROOM tackles a fundamental source of inefficiency in page loads that client-only solutions cannot address alone: VROOM decouples resource discovery from object evaluation (and thus, network delays from computational delays). By using server-provided hints, VROOM shifts resource discovery from being a client-only task to one in which the client and server cooperate.

## 9 Conclusions

The recognition that dependencies within the page load process are the dominant cause for slow page loads has led to a slew of solutions recently. However, all of these solutions either compromise security and privacy by relying on proxies to resolve dependencies or have limited ability to improve mobile web performance since they require clients to themselves discover the resources on any page. VROOM offers the best of both worlds: by having servers aid a client's discovery of resources (both via HTTP/2 PUSH and dependency hints), we decouple the client's processing and downloads of resources, but do so while preserving the end-to-end nature of the web. By improving CPU utilization, VROOM significantly decreases page load times compared to baseline HTTP/2.

## Acknowledgments

We thank the anonymous reviewers, our shepherd Mark Crovella, and Simon Pelchat for their valuable feedback on earlier drafts of this paper. This work was supported in part by a Google Faculty Research Award. Ravi Netravali's participation in the project was supported by NSF grant CNS-1407470 (PI: Hari Balakrishnan) and by the MIT Center for Wireless Networks and Mobile Computing.

## References

[1] Accelerated Mobile Pages Project. https://www.ampproject.org/.
[2] Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/.
[3] Google Developers - Simulate Mobile Devices with Device Mode. https://developers.google.com/web/tools/chrome-devtools/iterate/device-mode/.
[4] Google Page Speed. https://developers.google.com/speed/pagespeed/.
[5] H2O - The optimized HTTP/2 server. https://h2o.examp1e.net/configure/http2_directives.html#http2-casper.
[6] How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.
[7] HTTP Archive. http://httparchive.org/.
[8] HTTPS adoption *doubled* this year. https://snyk.io/blog/https-breaking-through/.
[9] Keynote: Mobile Commerce Performance Index. http://www.keynote.com/performance-indexes/mobile-retail-us.
[10] Latency Is Everywhere And It Costs You Sales - How To Crush It. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.
[11] Mobile Devices Now Driving 56 Percent Of Traffic To Top Sites. http://marketingland.com/mobile-top-sites-165725.
[12] The need for mobile speed: How mobile latency impacts publisher revenue. https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/.
[13] New findings: For top ecommerce sites, mobile web performance is wildly inconsistent. http://www.webperformancetoday.com/2014/10/22/2014-mobile-ecommerce-page-speed-web-performance/.
[14] nghttpx - HTTP/2 proxy. https://nghttp2.org/documentation/nghttpx-howto.html.
[15] Opera Mini & Opera Mobile browsers. http://www.opera.com/mobile/.
[16] Preload. https://www.w3.org/TR/preload/.
[17] SPDY. https://developers.google.com/speed/spdy/.
[18] visualmetrics. https://github.com/WPO-Foundation/visualmetrics.
[19] WPO Stats. https://wpostats.com/.
[20] M. Belshe, R. Peon, and M. Thomson. 2015. Hypertext Transfer Protocol Version 2. http://httpwg.org/specs/rfc7540.html.
[21] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *IMC*.
[22] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*.
[23] Abhijnan Chakraborty, Vishnu Navda, Venkata N Padmanabhan, and Ramachandran Ramjee. 2013. Coordinating Cellular Background Transfers using LoadSense. In *MOBICOM*.
[24] Jeff Erman, Vijay Gopalakrishnan, Rittwik Jana, and K.K. Ramakrishnan. 2013. Towards a SPDY'ier Mobile Web. In *CoNEXT*.
[25] Tammy Everts. 2013. Rules for Mobile Performance Optimization. *ACM Queue* 11, 6 (2013).
[26] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. 1999. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *SIGMETRICS*.
[27] David F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. 2004. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems* 5, 1 (2004), 1–28.
[28] Zhimei Jiang and Leonard Kleinrock. 1998. Web Prefetching in a Mobile Environment. *IEEE Personal Communications* 5, 5 (1998), 25–34.
[29] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. 2017. Improving User Perceived Page Load Times Using Gaze. In *NSDI*.
[30] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. 2010. WebProphet: Automating Performance Prediction for Web Services. In *NSDI*.
[31] Haohui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Pablo Montesinos. 2012. A Case for Parallelizing Web Pages. In *HotPar*.
[32] L. Meyerovich and R. Bodik. 2010. Fast and Parallel Web Page Layout. In *WWW*.
[33] David Naylor, Alessandro Finamore, Illias Leontiadis, Yan Grunenberger, Marco Mellia, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *CoNEXT*.
[34] Javad Nejati and Aruna Balasubramanian. 2016. An In-Depth Study of Mobile Browser Performance. In *WWW*.
[35] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *NSDI*.
[36] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *ATC*.
[37] Venkata N. Padmanabhan and Jeffrey C. Mogul. 1996. Using Predictive Prefetching to Improve World Wide Web Latency. In *SIGCOMM*.
[38] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Christopher Riederer. 2013. Give in to Procrastination and Stop Prefetching. In *HotNets*.
[39] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. 2017. Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance. In *WWW*.
[40] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. 2014. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *CoNEXT*.
[41] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *NSDI*.
[42] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How Speedy is SPDY?. In *NSDI*.
[43] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding Up Web Page Loads with Shandian. In *NSDI*.
[44] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why are Web Browsers Slow on Smartphones?. In *HotMobile*.
[45] Zhen Wang, Felix Xiazhou Lin, Lin Zhong, and Mansoor Chishtie. 2012. How Far Can Client-Only Solutions Go for Mobile Browser Speed?. In *WWW*.
[46] Zizhuang Yang. 2009. Every Millisecond Counts. https://www.facebook.com/note.php?note_id=122869103919.
[47] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. In *HPCA*.