# VROOM: Server-Aided Dependency Resolution for a Faster Mobile Web

Paper #163, 12 pages

*Abstract*— **The slowness of the web on mobile devices is a source of frustration for 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 address the impact of these dependencies on page load time, we present VROOM, a rethink of how clients and servers interact to facilitate web page loads. The key characteristic of VROOM is that, in contrast to the status quo where clients bear the onus of discovering the resources on any page, VROOM-compliant webservers aid clients in resource discovery. Thus, in any page load with VROOM, client-side processing of resources is largely decoupled from discovering and fetching resources, thereby enabling independent use of the CPU and the network. As a result, VROOM reduces the median page load time by XXX on popular News and Sports websites. To enable these benefits, our contributions lie in making webservers capable of aiding clients in discovering resources and judiciously scheduling client-side downloads of resources.**

## 1 Introduction

Despite web access on mobile platforms already constituting over half of the traffic to popular websites [12], web page loads on mobile devices remain woefully slow. Both industry reports [14, 11] and our own measurements (§2) show that a large fraction of mobile-optimized popular websites are much slower than user tolerance levels even on state-of-the-art mobile devices, e.g., the average web page takes 14 seconds to load on a 4G network [13]. The situation is unlikely to improve in the immediate future as trends indicate that faster cellular networks and CPUs will likely be offset by the increasing complexity of web pages. For instance, though the latency on LTE networks halved between 2013 [36] and 2016 [19], the average size of a web page roughly doubled during the same period [7].

Recent studies [41, 42] have found that dependencies between resources on a web page are a key reason why pages are slow to load. A mobile-optimized web page today includes roughly a hundred resources [6] on average, most of which a client browser discovers only after it has fetched and parsed/executed other resources, e.g., the client may learn that it needs to fetch an image only after executing a script which is discovered after parsing the page's HTML. Thus, after a page load begins, when the client can even discover the need to fetch a particular resource is limited by the network latency in fetching and the CPU's speed in parsing/executing other resources that appear earlier in the page load.

Prior work to address the impact of inter-resource dependencies on web performance has relied on offloading dependency resolution—either to remote proxies [40, 35, 22, 43] or to the domain serving the root HTML for a page [34]—in order to benefit from faster CPUs and connectivity. However, when clients load pages via remote proxies, several challenges arise: clients have to trust that the security of HTTPS content has been preserved; indirect client-server communication via a proxy can potentially degrade performance [39]; and setting up a scalable fault-tolerant proxy that can serve all page loads for all users is no mean feat. Moreover, to preserve web providers' ability to personalize content, clients have to share their cookies for all domains with the remote entity performing dependency resolution on their behalf.

Given these limitations of prior solutions in tackling the adverse impact of dependency resolution on the performance of web page loads, with VROOM, we pursue an end-to-end redesign of how browsers and webservers interact. We pursue this direction inspired by the success of SPDY, which though initially used only between the Chrome browser and Google's webservers is now incorporated into version 2 of the HTTP protocol, support for which is steadily increasing [8].

Our primary insight in designing VROOM is to decouple the two purposes of client-side processing of resources when loading a web page: 1) discovering resources to fetch, and 2) figuring out how to collectively use the resources to render the page. Decoupling these maximizes the efficiency of page loads because fetching all the resources on a page is constrained by the network whereas the CPU limits the parsing and execution of the fetched resources. Our high-level approach to facilitate this decoupling is to have webservers aid clients in resource discovery, in contrast to the status quo where clients bear all the onus of identifying which resources to fetch to load a page. Web providers have an incentive to incur the associated resource overheads because this helps improve user experience on their site, thereby increasing revenue. We make three contributions in designing VROOM to realize this approach.

First, to enable webservers to aid clients in resource discovery, we revise the interaction between them. In addition to returning any requested resource, VROOM-compliant webservers not only *push* the content of dependent resources but also return *dependency hints* in the form of URLs of resources that the client should fetch. VROOM's use of dependency hints is key because leaving it to clients to fetch dependent resources hosted in other domains lets clients verify the integrity of HTTPS content and preempts the need for them to share their cookies for one domain with other domains. As a result of this input from servers, a client's ability to discover the set of resources that it should fetch to load a page is no longer constrained by the speed with which it can process page 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 webservers 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 [40, 35] or offline [34, 22] dependency resolution, we show how to *combine* the two to accurately identify the set of resources that a client will need to fetch within a specific page load. Critically, our design ensures that servers can recognize and avoid sending dependency hints for intrinsically unpredictable resources—ones which vary even across back-to-back loads of the page—and for resources subject to personalization, so that a client can discover these resources either on its own or from other domains, rather than incur the overhead of fetching resources that are unnecessary for its page load.

Lastly, while the additional input from VROOM-compliant webservers reduces the latency for clients to discover all resources on a web page, client browsers still need to parse and execute the fetched content, which we (§2) and others [33] find to be the primary bottleneck in mobile page loads. To maximize CPU utilization, we leverage the property that resources that need to be parsed/executed constitute only a quarter of the bytes on the average mobile web page [6]. We coordinate server-side pushes and client-side fetches such that these resources are fetched preferentially over others; as a result, while the processing of these resources keeps the CPU busy, the client can utilize its network to fetch other resources on the page in parallel.

Our implementation of VROOM enables page loads in Chrome from the Mahimahi [35] 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 seconds to YYY 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 YYY% 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 on mobile devices, show that state-of-the-art protocols are insufficient, and estimate the potential to reduce page load times.

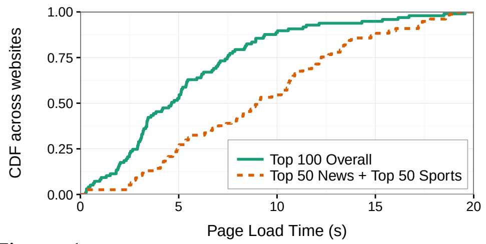


Figure 1: **Page load times on today's mobile web.**

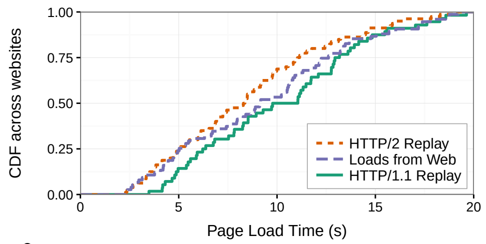


Figure 2: **Estimation of page load time improvements that would be enabled once HTTP/2 is globally adopted.**

**Problem: Poor load times.** We demonstrate the slowness of the mobile web on two sets of websites: the 100 most popular websites and the top 50 sites each from the News and Sports categories, all based on Alexa's ranking of websites in the US; our focus in this paper is on websites that serve relatively static content, unlike webmail and social networks, but whose pages do change over time and are customized based on user profiles. We study popular sites because they are more likely to have applied known best practices such as minifying JavaScripts and eliminating HTTP redirects [4]. For each site, we load the landing page on a Nexus 6 smartphone connected to Verizon's LTE network with excellent signal strength and consider the median page load time[1] across 5 loads.

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 for which a typical user is willing to wait [25, 32, 17]. When considering News and Sports sites, which are more complex than the average site [21], the load time for the median site is even higher at roughly 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.

**HTTP/2 is insufficient.** To estimate the potential impact of the adoption [8] of the recently standardized version 2 [10] of HTTP, we use Mahimahi [35] to record the content of pages and replay them in two environments where either HTTP/1.1 or HTTP/2 is universally used; more details about our replay setup are in Section 7. Figure 2 compares the distribution of load times in these two cases with the load times observed in practice. First, given that the adoption of HTTP/2 is still in its nascency, the fidelity of our replay setup is confirmed by the close

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

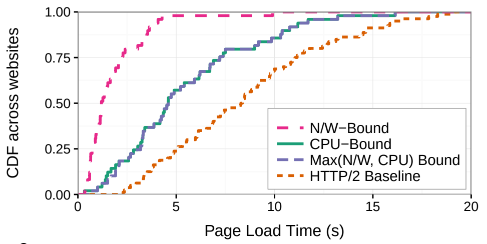


Figure 3: **Potential for reducing page load times by making better use of the client's CPU and network.**

match between load times measured when loading pages from real webservers and when replaying the loads with all domains using HTTP/1.1. Second, our results portend that, once HTTP/2 is universally adopted, the median page load time across popular News and Sports websites will drop to roughly 8 seconds, which is still significantly higher than users' tolerance levels. Note that we do not leverage HTTP/2's server PUSH capability, which enables servers to push resources that have not yet been requested, because it is challenging to determine which resources should be pushed, as we will discuss later.

**Cause: Poor CPU/network utilization.** Though HTTP/2's multiplexing of multiple transfers on the same TCP connection reduces connection setup overheads, the root cause for the slowness remains: because 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 a HTML/CSS or execute a JavaScript 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 this resource by parsing/executing some other resource. Thus, as shown by the example in Figure 4(a), both the CPU and the network end up limiting each other's utilization.

We estimate the potential gains from a redesign of the page load process that would fully utilize at least one of the client's CPU or access link bandwidth on every page load. To mimic a setting where the network bandwidth is the bottleneck, we replay every page load after replacing the HTML with one which links to all the resources on the page, so that the browser fetches all resources from parsing the HTML. We replace every resource on the original page with a blob of the same size and embed each blob in a separate iframe such that the browser downloads every blob but does not attempt to interpret its content. Whereas, to emulate a setting where the client's CPU is the bottleneck, we load every page in our replay setup with the client phone connected to WiFi and with the replay server hosted within a 1ms delay from the client.

Figure 3 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. If we consider the higher of the CPU-bound and network-bound estimates for every page, we see that the CPU is typically the bottleneck in mobile page loads, as also observed by other recent studies [33]; page load time for the median web page reduces from 8 seconds with HTTP/2 to 4.5 seconds. These results highlight the utility of redesigning the page load process such that usage of the client's CPU and access link are decoupled, thereby utilizing at least one of them to capacity.

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

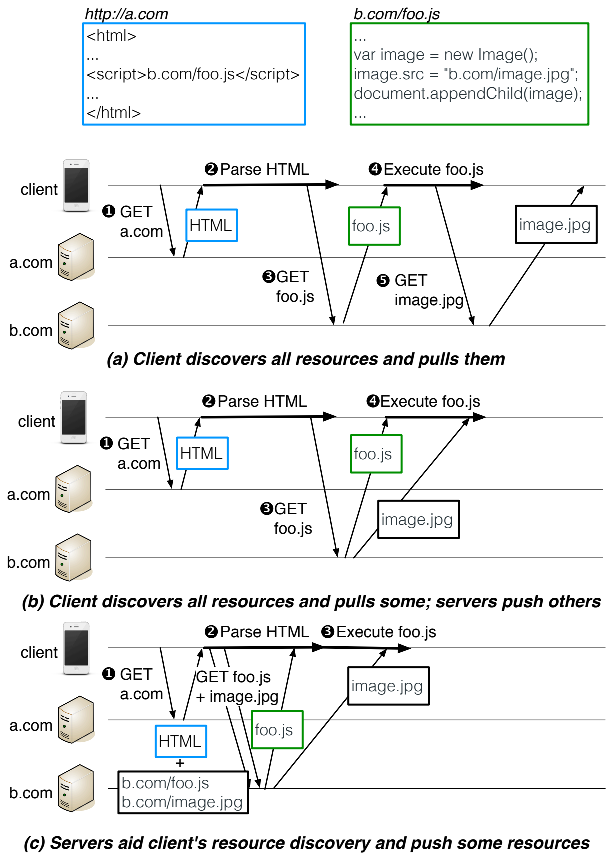
- Page load times even for popular websites are currently significantly higher than tolerable, particularly for sites in categories that have more complex web pages than others.
- The reduction in load times that we can expect from the adoption of HTTP/2 will not suffice.
- However, we could potentially halve the median load time if the page load process were redesigned to more efficiently use the client's CPU and network.

## 3 Approach

To minimize page load times by decoupling use of the client's CPU and network, and thereby maximizing the utilization of either, web page loads would ideally work as follows. When a client browser seeking to load a particular web page issues a request for the page, it would receive back *all* the resources needed to render the page, not only 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 the page content at once in response to one request would maximize the utilization of the client's access link bandwidth and eliminate the need for repeated latency-onerous interactions between the client and webservers. 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 while fetching resources in parallel.

### 3.1 Limitations of HTTP/2 PUSH

It may appear that such an ideal design of the page load process is indeed feasible today thanks to HTTP/2-compliant webservers being capable of *pushing* resources to clients [10]. However, the typical characteristics of web pages 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 [9]; and 3) page content is increas-



http://a.com

```
<html>
...
<script>b.com/foo.js</script>
...
</html>
```

b.com/foo.js

```
...
var image = new Image();
image.src = "b.com/image.jpg";
document.appendChild(image);
...
```

Figure 4: **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 content: (a) 5 steps 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 steps on critical path with CPU use blocking use of the network in step 2 and vice-versa in step 3, (c) 3 steps on critical path with CPU and network utilized throughout.**

ingly personalized—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 from other domains. If webservers were to fetch resources from external domains and push them to clients, this would prevent clients from verifying the integrity of secure page content. Moreover, since any client's request to a webserver will only include the client's cookie for that domain, resources fetched from other domains by this webserver will not reflect any personalization of content by those domains.
- Webservers in every domain could push all locally hosted resources relevant to a page load when the client issues a request to that domain. But, the client will discover resources that it needs to fetch from other domains only after processing the resources received from the domain hosting the root HTML for the page, thus making the CPU a potential bottleneck in the client's fetching of resources.
- During a page load, if every domain independently pushes its resources on the page to the client, the client's receipt from one domain of a resource that it needs to process (i.e., HTML, CSS, or JavaScript) can be delayed due to sharing of the access link's bandwidth with other resources (such as bulky images) from other domains, thus making 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 *only* on webservers pushing content to optimize page loads, we rely on an additional server-side capability. When a webserver receives a request for a resource, apart from pushing the content for other dependent resources, the server also has the option of returning a list of URLs for dependent resources, which we refer to as *dependency hints*. For example, webservers can include such a list of URLs as an additional header in HTTP responses. When a client receives dependency hints in response to one of its requests, it can fetch every resource whose URL is included in the list, without having to discover this resource on its own by processing other resources on the page. In fact, when a HTTP response includes a list of URLs in the form of a Link Preload [18] header, some legacy browsers already include support[3] to fetch the included URLs immediately upon receipt of the HTTP response's headers.

In response to a request for a resource, having webservers send clients the URLs for dependent resources, rather than the content of these resources as done with PUSH, offers several advantages.

- Any webserver can safely send clients dependency hints for third-party resources. For any URL received via dependency hints, a client will fetch it from the respective domain, thereby being able 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, thus making it easier to minimize bandwidth waste compared to content push.
- The client maintains control over its concurrent fetches of resources from multiple domains; it can

[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 webservers, e.g., in a cookie [5].

[3]https://www.chromestatus.com/feature/5757468554559488

coordinate its downloads so that the resources that it needs to process are not delayed.

## 4 Design

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

- In response to a request from a client, how can a webserver accurately identify the list of dependent resources that it should inform the client about, including ones hosted in other domains, without getting clients to waste bandwidth downloading resources irrelevant to the ongoing page load?
- When informing clients about dependent resources, how should a webserver decide whether it should or should not return dependencies derived from external resources given that it lacks the user's cookies for other domains and hence cannot account for their personalization of content?
- When during the page load should clients fetch the URLs included in dependency hints received from webservers, and should every webserver push all the resources it can?

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, and no single developer is likely to have complete knowledge about all dependencies on a page; 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 webservers of that domain.

Figure 5 illustrates the server-side and client-side components of VROOM, which we describe next.

### 4.1 Server-side dependency resolution

First, when a client loads a web page on the site foo.com, let us consider a webserver in this domain that receives a HTTP request for the HTML of the page. Both to push resources and to return dependency hints for resources the client should fetch, the server needs to identify other resources on the page.

**Strawmans for resource discovery.** To see the challenge in server-side resource discovery, consider the following two strawman approaches.

On the one hand, when a webserver receives a request for the HTML of a page, the server could itself load the page on the fly, mimicing a client browser, to identify other resources on the page. However, many of the URLs fetched when loading the page at the server will not be requested as part of the page load process at the client: 1) even back-to-back loads of a page often differ in the set of URLs fetched (e.g., ads and analytics providers typically insert a randomly selected identifier into the URLs they fetch), and 2) loading a page at a webserver cannot account for the personalization performed by other domains, since the server only has the user's cookie for its own domain. If webservers fail to account for these discrepancies and either push to the client or ask the client to fetch URLs that do not contribute to its page load, the user is likely to experience a *higher* load time with VROOM, instead of benefiting from it.

Alternatively, for every page for which it serves the HTML, a webserver can periodically load the page in the background. This enables the server to account for non-determinism; whenever a client does request for the HTML, the server can return the set of resources that it has repeatedly observed on recent loads of the page. In this case, we risk missing a large fraction of URLs that a client will need to fetch when loading the page; given the rate of flux in page content, a significant fraction of URLs seen on a page even an hour ago are often no longer on the page when serving a client's request [22].

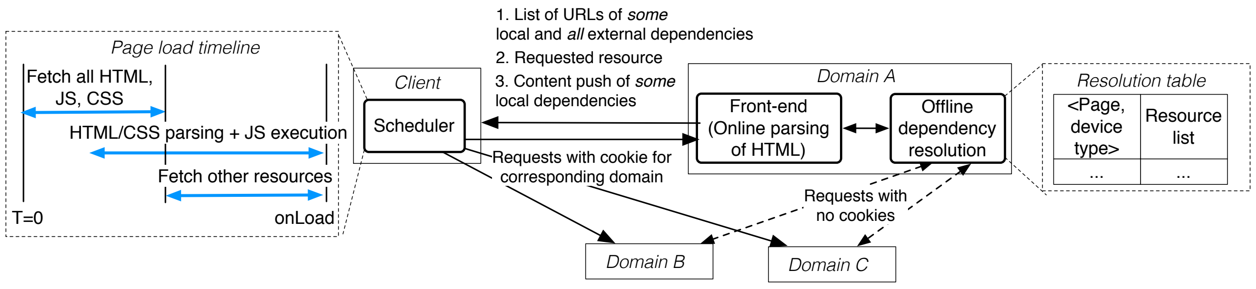
**Offline + online resource discovery.** The two strawman approaches for server-side resource discovery illustrate the following trade-off: VROOM must ensure that a client does not end up fetching resources that are not necessary for the page load, as this may inflate page load times, but if we are too conservative and let the client discover on its own many of the URLs that it needs to load the page, the utility of server-aided resource discovery will be minimal.

To address this trade-off, we observe that *both* offline and online dependency resolution are necessary at the server. Periodic offline resolution of a page helps confirm which URLs are consistently fetched when loading the page. Whereas, online resolution helps account for flux in page content. Next, we describe how VROOM leverages the strengths of both approaches.

#### 4.1.1 Offline dependency resolution

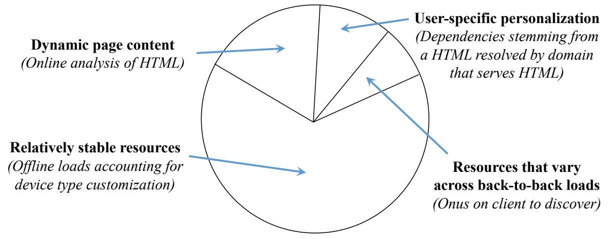
Offline server-side determination of the resources on a page works as described previously: the page is loaded periodically (once every hour in our implementation), and at any point in time, URLs fetched in all recent loads are considered likely to also be fetched when a client loads the page.

However, even the largely stable subset of resources on a page can vary across different types of client devices. For example, it is common for website providers to use CSS stylesheets and JavaScripts that cause different clients to fetch images of different sizes on the same page depending on the client's display resolution or pixel density. Figure 7 shows that the stable set of URLs fetched across multiple loads of the same page can differ significantly across devices.



Page load timeline
Fetch all HTML, JS, CSS
HTML/CSS parsing + JS execution
Fetch other resources
T=0
onLoad
Client
Scheduler
1. List of URLs of *some* local and *all* external dependencies
2. Requested resource
3. Content push of *some* local dependencies
Requests with cookie for corresponding domain
Domain A
Front-end (Online parsing of HTML)
Offline dependency resolution
Resolution table
<Page, device type>
Resource list
...
...
Requests with no cookies
Domain B
Domain C

Figure 5: **Illustration of the components in VROOM and the interactions between them.**



Dynamic page content
(Online analysis of HTML)
User-specific personalization
(Dependencies stemming from a HTML resolved by domain that serves HTML)
Relatively stable resources
(Offline loads accounting for device type customization)
Resources that vary across back-to-back loads
(Onus on client to discover)

Figure 6: **Breakdown of the types of resources on any web page, and summary of techniques used in VROOM for server-side dependency resolution to account for each type of resource.**

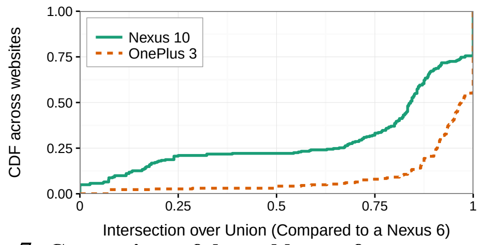


Figure 7: **Comparison of the stable set of resources on each page when the user device is a Nexus 6 compared with when the user device is a Nexus 10 or a OnePlus 3.**

VROOM's offline server-side dependency resolution efficiently accounts for device-specific customization of resources in two ways. First, for every page that the server loads periodically, the server need not load the page once for each device type; 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 7, 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 perform the offline page loads leveraging device emulation supported by browsers such as Chrome [3].

### 4.1.2 Online HTML analysis

While offline dependency resolution helps identify the stable set of resources on a page, relying solely on offline loads will fail to account for flux in page content. For example, the set of stories or set of products on the landing page of a news or shopping site change often.

To account for dynamic content, when VROOM-compliant webservers respond to a request with a HTML, they not only inform the client of dependencies discovered from loading this HTML offline, but also include all URLs seen in the HTML 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 7 that accounting for the URLs in HTMLs suffices on most pages to capture the flux in page content, while also ignoring the unpredictable resources (i.e., ones which vary even across back-to-back loads) that clients must discover on their own. Importantly, we find that server-side parsing of HTMLs as they are being served adds a median delay of only roughly 100ms across the landing pages of the top 1000 websites, an overhead that clients can afford to bear in order to avail the multi-second reduction in page load times made possible by server-aided resource discovery.

### 4.1.3 Accounting for personalization

As discussed earlier (Section 3), unlike content push, dependency hints allow a webserver to inform clients even of dependent resources served by other domains. Therefore, the webserver that serves the root HTML of a page can include with its response the URLs of *all* resources identified via offline and online analysis. However, doing so fails to account for personalization of resources; the set of URLs that a client will end up fetching, especially ads, will likely differ from the URLs fetched during server-side dependency resolution.

One way to deal with personalization would be that, when a webserver responds to a request, it never returns dependencies that are derived from external resources. In other words, any resource that is discovered during the page load by parsing or executing an external resource is

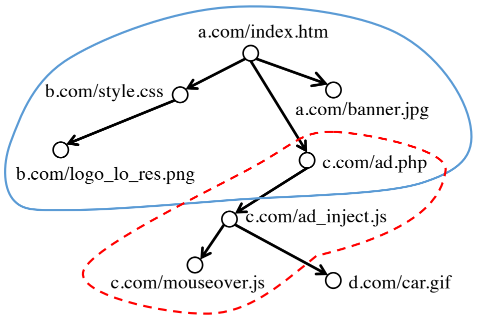


Figure 8: **Illustration of how VROOM-compliant servers account for personalization. A client that requests for index.htm from a.com only discovers the resources within the solid blue envelope, because a request for ad.php returns a HTML, whose content could be personalized to a specific user. The client 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.**

deemed as one that could differ due to personalization. Accounting for personalization in this manner would however inflate the latency incurred by the client in discovering all resources on the page, thus limiting its ability to fully utilize the network. For example, in Figure 8, though a.com will inform a client about b.com/style.css in response to a request for index.htm, the client will have to discover the need to fetch b.com/logo_lo_res.png only when it requests style.css from b.com.

To reduce the latency of discovering dependencies from webservers, we observe that websites are personalized primarily in two ways: by customizing the content of HTML responses,[4] and by adapting the execution of scripts. Server-aided resource discovery in VROOM accounts for these two types of personalization as follows. First, when a webserver receives a request for a page from a client, the client's discovery of dependencies stemming from external resources are deferred to the respective domain only when this external resource is a HTML (e.g., an embedded iframe); dependencies derived from other types of external resources (e.g., style.css in Figure 8) are not deferred. Second, resources that are determined based on user-specific adaptation in JavaScript execution are left to clients to discover on their own. Typically, JavaScript-based personalization will vary over time, and hence, such resources will get filtered out from the stable set of resources identified via offline dependency resolution.

In summary, Figure 6 lists the various techniques used for server-side dependency resolution in VROOM, in order to account for different types of resources on any

[4]While the content of resource types other than HTML (i.e., CSS, scripts, images, and videos) could also be user-specific, this seldom occurs in practice. Customization of these other types of resources is at best device-specific, which we previously accounted for.

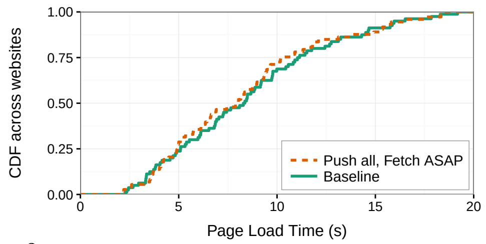


Figure 9: **Comparison of page load times with baseline HTTP/2 versus when all servers push any resources they can and clients immediately fetch URLs received via dependency hints.**

page. As we show later in Section 7, in combination, these techniques are able to accurately identify the URLs that a client will need to fetch to load a page, save for those URLs that vary even between back-to-back loads.

## 4.2 Cooperative request scheduling

Next, we turn our attention to questions that must be answered for clients to benefit from server-side discovery. How should webservers combine the use of content 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 webserver receives a client's request for a HTML, out of all the dependent resources that the server wishes to inform the client about, the server can push to the client the subset of resources that it can, i.e., all resources served from the same domain; clients will not accept pushes of resources from other domains. The server can inform the client of all other dependencies via dependency hints, i.e., by augmenting its response with a list of URLs. As soon as the client receives this list, it can initiate downloads for all URLs in the list.

**Problem: Contention for bandwidth slows down processing.** Though this simple strategy for server-aided dependency resolution significantly reduces the time by when the client discovers resources on the page and begins fetching them, the improved utilization of the client's access link comes at the expense of slowing down the client's processing of resources on the page. This is because, for many of the resources on the page, the client begins receiving them sooner than it would when discovering resources on its own. As a result, among the resources that need to be processed, downloads of those fetched early in the page load get slowed down contending for bandwidth with the greater number of concurrent downloads. The resulting under-utilization of the CPU is significant because, as shown by prior work [33] and confirmed in Section 2, the CPU is a bigger bottleneck in mobile page loads than the network. Consequently, Figure 9 shows that we see no improvements in page load times when we apply the above-described straightforward strategy to benefit from server-side resource discovery.

**Solution: Prioritization via selective push and deferred downloads.** The solution to this problem is to prioritize the fetches of resources that need to be processed (HTML, CSS, and JS) over ones that need not (e.g., images and videos). This classification of resources into high and low priority 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 during 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, e.g., HTTPArchive shows that images and videos account for almost 75% of the bytes on average on a mobile web page.[5]

This prioritization of resources that need to be processed is achieved in VROOM via two means. First, when a webserver responds to a client's request for a HTML, out of all the dependencies the server identifies, it pushes the content of all 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. Once resource discovery from servers is complete and the client has finished fetching all the high priority resources discovered, it issues requests for all other resources at once. Though the prioritization could be more fine-grained, e.g., prefer fetching resources that need to be processed earlier in the page load over those processed later, but we find that two levels of priority (based on whether a resource will be processed) suffices, as we show in Section 7.

## 5 Implementation

Realization of VROOM requires both server-side (dependency resolution; pushing resources and sending dependency hints to clients) and client-side (scheduling fetches of resources discovered) changes. Since this preempts evaluation in the wild, we have implemented VROOM to facilitate web page loads in Chrome from the Mahimahi [35] replay environment.

### 5.1 Server-aided resource discovery

VROOM-compliant servers inform clients of dependent resources via content pushes and dependency hints. For the former, we leverage HTTP/2's PUSH capability; since Mahimahi runs Apache webservers, which does not yet support HTTP/2, we run an instance of nghttpx [15], a HTTP/2 reverse proxy in front of every webserver. For the latter, we rely on embedding additional headers in HTTP responses. For example, when a HTTP response includes a URL specified via the "Link preload" header [18], which is currently supported only in Chrome,[6] the browser will immediately issue a request for this URL upon receipt of the header.

[5]http://mobile.httparchive.org/interesting.php

### 5.2 Scheduling requests with JavaScript

To schedule client-side downloads of URLs learned via dependency hints (Section 4.2), we use a JavaScript-based request scheduler. Once we record a page in Mahimahi, we modify the page's top-level HTML using BeautifulSoup [2] to add our scheduler script as the first tag in the HTML, thereby ensuring that the browser executes this script as soon as it begins parsing the HTML. When it begins execution, our scheduler script performs two steps. First, it defines an `onload` handler, *response_handler*, which it attaches to each `XMLHttpRequest` (XHR) that it makes; this handler fires every time the browser receives a response. Second, the script issues an XHR for the page's HTML, whose URL we embed as an attribute in the `<html>` tag.[7]

Thereafter, our scheduler runs in an event-driven loop. Whenever the browser receives a response and invokes our *response_handler*, we inspect the headers included in the response,[8] issue XHR requests for the high priority dependency hints (specified via the Link preload header), and record the hints for low priority resources (specified via a custom header). Once all high priority resources learned via dependency hints have been received, the handler issues requests for all the low priority resources discovered.

It is important to note that 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.

By using a JavaScript-based request scheduler, our implementation of VROOM can accelerate page loads on unmodified commodity browsers. However, JavaScript's event-driven model is constrained by the language's single-threaded nature, i.e., handlers added to the event queue may not be executed while other scripts on the page are being processed. As a result, response_handlers may not fire as soon as responses arrive to the browser, potentially delaying the fetches for lower priority re-

[6]Other browsers are in the process of adding support for Link preload: https://bugzilla.mozilla.org/show_bug.cgi?id=1222633

[7]Note that the scheduler removes this attribute and its own DOM node from the page once the XHR for the top-level HTML is issued; this ensures that subsequent accesses to the DOM are not affected.

[8]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 header 'x-vroom-low-priority'.

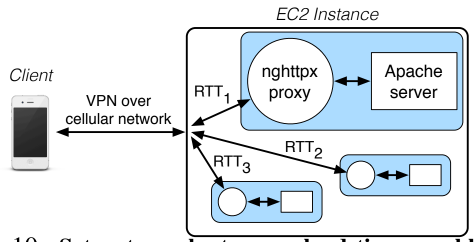


Figure 10: **Setup to evaluate page load times enabled by our implementation of VROOM.**

sources. Therefore, in the future, incorporation of the scheduling logic into the browser would be desirable.

## 6 Optimizations

- First

## 7 Evaluation

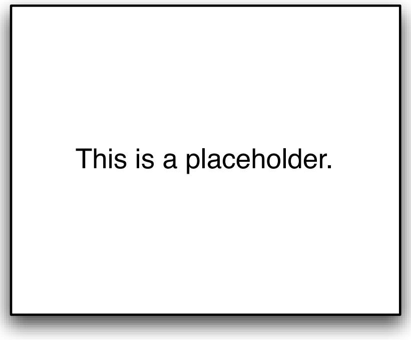
We evaluate VROOM from two perspectives: 1) the accuracy with which servers can aid resource discovery for clients, and 2) the performance benefits for users.

### 7.1 Impact on client performance

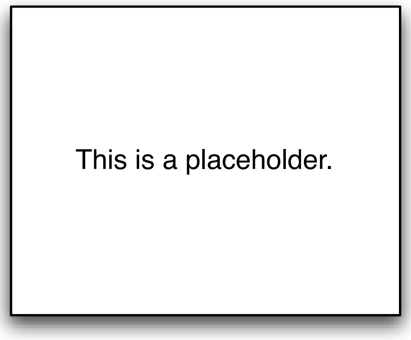
**Setup.** We use the setup shown in Figure 10 to experimentally evaluate our implementation of VROOM. We load pages in Chrome for Android on a Nexus 6 smartphone connected to a Verizon LTE hotspot. The phone is also connected via USB to a desktop, which subscribes to events exported by Chrome via the Remote Debugging Protocol (RDP). The phone has a VPN tunnel setup to a c4.xlarge EC2 instance (in a data center that is 25ms RTT away), on which we host Mahimahi [35]. For every web page on which we test VROOM, we initially load the page via the EC2 instance 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 webservers. Thereafter, when replaying page loads, we configure Mahimahi such that any traffic between the phone and one of the webservers is subjected to not only the delay over the cellular network but also the median RTT observed between the EC2 instance and the corresponding webserver when recording page contents.

We evaluate the utility of VROOM on the landing pages of the top 50 News and top 50 Sports websites;[9] 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 seconds. We load each page 5 times in our replay setup and consider the load with the median load time. During these loads, webservers 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.

[9]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)

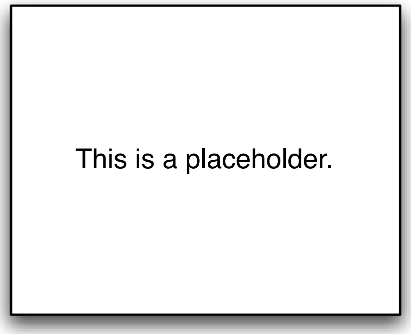


(b)

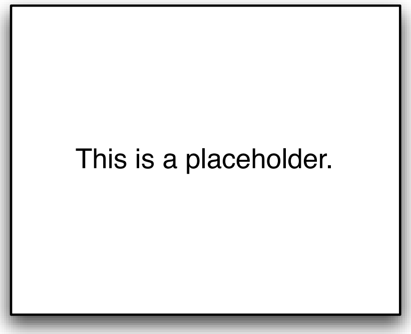
Figure 11: VROOM **yields significant reductions compared to baseline HTTP/2 with respect to (a) page load times and (b) time by when above-the-fold rendering completes.**

**Improvement in page load times.** First, we compare page load performance when using VROOM with that when doing a normal HTTP/2 based replay (i.e., same setup as Figure 10, except that servers simply return requested resources), which we refer to as the baseline.

**Latency in discovering and fetching resources.** A key reason for the page load time improvements with VROOM is that server-aided resource discovery enables clients to discover resources and complete fetching them much sooner than in normal page loads. Figure 12 depicts these improvements both when considering all resources on each page, and also when considering only the high priority resources (i.e., HTML, CSS, and JS objects, which are the ones need to be parsed or executed). Reducing the time by when the client completes fetching all resources on the median page from XXXs to YYYs is obviously critical because no further network activity is necessary to complete the page load. But, in addition, the drop in the time by when all high priority resources finish downloading—the median reduces from XXX with the baseline to YYY with VROOM—is also critical since utilization of the CPU and the network are decoupled thereafter; none of the resources that subsequently fin-



(a)



(b)

Figure 12: **In comparison to baseline HTTP/2, VROOM reduces the latency in both (a) discovering resources and (b) completing their downloads.**

ish downloading need to be processed. Both of these improvements are made possible because of the speedup in the client discovering resources: in the median, a XXX% drop in the latency to discover all resources and a YYY% drop in discovering all high priority resources.

**Utility of scheduling.** Beyond faster resource discovery, performance improvements with VROOM also critically rely upon judicious pushes and downloads of discovered dependencies. As shown earlier in Section 4.2, naively pushing resources whenever a server can and fetching a resource as soon as the client discovers it does not help reduce page load times.

Figure 13 illustrates the utility of the cooperative scheduling in VROOM. Because only high priority resources are pushed by servers and are preferentially fetched by clients, we see that the time by when the downloads of all important resources complete significantly reduces with the use of scheduling compared to without; on the median web page, a decrease from XXXs to YYYs. This helps increase the CPU utilization as the browser has to spend lesser time waiting to receive resources that it needs to process.

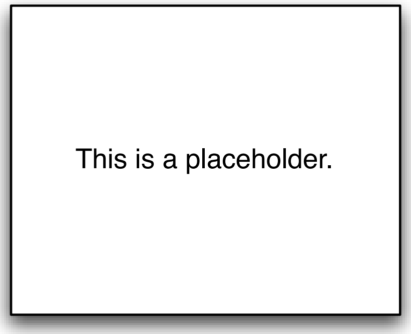


Figure 13: **In VROOM, judicious scheduling of resources results in significantly faster completion of the receipt of resources that need to be processed.**

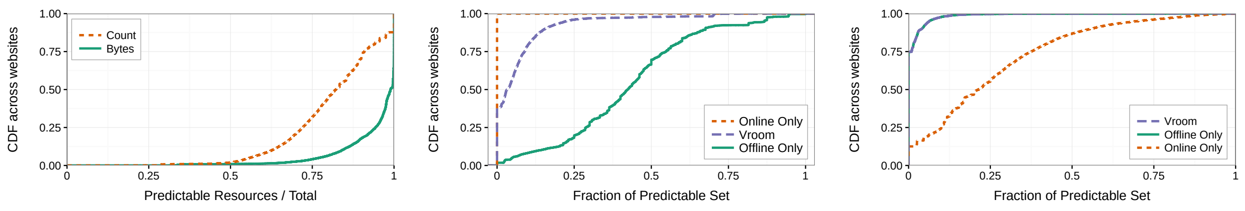
### 7.2 Accuracy of server-side dependency resolution

**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 listing 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. In fact, 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. For the offline dependency resolution, we load every page once every hour. For online analysis, we model the fact that the server serving any HTML on a page—either the root or one embedded in an iframe—to a client can analyze 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 (Section 4.1.3).

**Strategies for server-side resource discovery.** Recall that our design requires VROOM-compliant webservers to combine both offline discovery and online analysis to identify dependencies in a manner that can cope with flux in page content, is robust to non-determinism, and accounts for personalization. We compare our design 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



(a) (b) (c)

Figure 14: **(a) Among the URLs of dependent resources derived from the HTML of a page, except for those dependent on 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.**

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 14(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 other embedded HTMLs), 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—both computed as fractions of the predictable subset.

**Results.** First, Figure 14(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 14(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 ends up identifying many extra resources, which inflate the predictable subset by as much as 20% in the median case.

## 8 Discussion

**Deployability.** Unlike attempts at clean-slate redesigns of the Internets architecture, redesigning client-server interactions on the web has been shown to be deployable 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 webservers, unlike an ISP having to rely on other ISPs to forward traffic, 2) a few popular browsers and webservers 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.

**Efficiency.** As most popular websites host thousands of web pages, loading each of these every hour to facilitate offline dependency resolution will likely be onerous. However, 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.

## 9 Related work

Much prior work has focused on understanding and improving mobile web performance. Here, we highlight areas of related work that influenced the design of VROOM.

**Mobile web performance:** Prior measurement studies [44, 33] 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 (thereby avoiding render blocking), and uses a CDN—run by Google—to serve AMP-enabled content with HTTP/2. VROOM is complementary to AMP and can further improve performance by issuing asynchronous fetches earlier using server-provided hints.

**Dependencies in page loads:** Recent systems [22, 34, 27, 21] use offline analysis to discover dependencies inherent to web pages. These systems then leverage a priori knowledge of inter-object dependencies to issue requests in a way that improves performance. However, due to the steady flux in web content, previously generated dependency graphs can only capture the structure of a page, but not the exact set of resources (i.e., URLs) needed to load it. 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 [40, 35, 43, 22, 16]. However, the reliance on web proxies poses several drawbacks: proxies must be scalable and fault-tolerant to serve large user populations, leakage of user privacy is a concern, and indirect routing through a proxy can actually degrade performance when the client has a fast network connection [39].

**Network optimizations for faster page loads:** HTTP/2 [20] (formerly SPDY [30]) 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). 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 [23]. 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 [28, 26, 24, 37, 45, 28]. 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 [29, 31] and improved hardware [46]. 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.

## 10 Conclusions

The recognition that dependencies within the page load process is the dominant cause for slow page loads has led to a slew of proxy-based solutions [40, 35, 22, 43] recently. In contrast, we present a more scalable and secure alternative: clients interact with webservers directly, but servers do more than simply return requested resources. By judiciously combining the use of HTTP/2 PUSH and dependency hints via HTTP headers, VROOM enables servers to aid clients in resource discovery, thereby decoupling the client's processing and downloads of resources. Consequently, the improved CPU and network utilization enables significant improvements in page load times compared to what we can expect once HTTP/2 is universally adopted.

## 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] HTTP archive mobile. http://mobile.httparchive.org/.

[7] HTTP Archive: Trends. http://httparchive.org/trends.php?s=All&minlabel=Feb+1+2013&maxlabel=Feb+1+2016.

[8] HTTP/2 dashboard. http://isthewebhttp2yet.com.

[9] HTTPS adoption *doubled* this year. https://snyk.io/blog/https-breaking-through/.

[10] Hypertext transfer protocol version 2 (HTTP/2). RFC 7540.

[11] Keynote: Mobile retail performance index. http://www.keynote.com/performance-indexes/mobile-retail-us.

[12] Mobile devices now driving 56 percent of traffic to top sites. http://marketingland.com/mobile-top-sites-165725.

[13] The need for mobile speed: How mobile latency impacts publisher revenue. https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/.

[14] 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/.

[15] nghttpx - HTTP/2 proxy. https://nghttp2.org/documentation/nghttpx-howto.html.

[16] Opera Mini & Opera Mobile browsers. http://www.opera.com/mobile/.

[17] Page speed: Are slow loading pages killing your growth? https://www.shopify.com/enterprise/60726275-page-speed-are-slow-loading-pages-killing-your-growth.

[18] Preload. https://www.w3.org/TR/preload/.

[19] T-Mobile ties Verizon in US-wide speed test but lags in total coverage. http://arstechnica.com/business/2016/02/t-mobile-ties-verizon-in-us-wide-speed-test-but-lags-in-total-coverage/, 2016.

[20] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. http://httpwg.org/specs/rfc7540.html, May 2015.

[21] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *IMC*, 2011.

[22] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI*, 2015.

[23] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier mobile web. In *CoNEXT*, 2013.

[24] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *SIGMETRICS*, 1999.

[25] D. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 2004.

[26] Z. Jiang and L. Kleinrock. Web prefetching in a mobile environment. *IEEE Personal Communications*, 5(5):25–34, 1998.

[27] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating performance prediction for web services. In *NSDI*, 2010.

[28] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant web browsing for mobile devices. In *ASPLOS*, 2012.

[29] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *HotPar*, 2012.

[30] The Chromium Projects. SPDY. https://www.chromium.org/spdy, 2015.

[31] L. Meyerovich and R. Bodik. Fast and parallel web page layout. In *WWW*, 2010.

[32] F. Nah. A study on tolerable waiting time: How long are Web users willing to wait? *Behaviour & Information Technology*, 23(3), May 2004.

[33] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *WWW*, 2016.

[34] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, 2016.

[35] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, 2015.

[36] OpenSignal. The state of LTE. https://opensignal.com/reports/state-of-lte/, 2013.

[37] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[38] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to procrastination and stop

prefetching. In *HotNets*, 2013.

[39] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-aware compaction for accelerating mobile web transfers. In *MobiCom*, 2015.

[40] A. Sivakumar, S. P. N., V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy assisted browsing in cellular networks for energy and latency reduction. In *CoNEXT*, 2014.

[41] X. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *NSDI*, 2013.

[42] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[43] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with Shandian. In *NSDI*, 2016.

[44] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *HotMobile*, 2011.

[45] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *WWW*, 2012.

[46] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.