MetaPush: Cellular-Friendly Server Push For HTTP/2

Bo Han[*]
AT&T Labs – Research
Bedminster, NJ
bohan@research.att.com

Shuai Hao[*]
AT&T Labs – Research
Bedminster, NJ
haos@research.att.com

Feng Qian[*]
Indiana University
Bloomington, IN
fengqian@indiana.edu

ABSTRACT

MetaPush is a novel server push framework aiming at reducing web page load time. The key idea is to strategically leverage Server Push, a built-in feature in HTTP/2, to preemptively push web pages' metadata, which can later be leveraged by the client to early-fetch critical resources. We demonstrate MetaPush is particularly suitable for cellular networks, providing negligible bandwidth overhead (around 0.4%), improved page load time (up to 45%), and reduced radio energy utilization (up to 37%) over HSPA+ networks.

CCS Concepts

•Networks → Application layer protocols;

1. INTRODUCTION

Modern web pages are rich and complex, even for their mobile versions [17]. Loading a web page thus involves complex interactions among multiple entities including the network, cache, page parser, JavaScript/CSS evaluator, and rendering engine [20]. Accordingly, numerous optimization techniques have been proposed to improve various aspects in this sophisticated system. For example, CDNs are deployed to reduce network latency, pages are rewritten to reduce the parsing time [1], multiplexing is introduced to optimize transport-layer performance [3, 7], and compact image formats [8] are used to reduce content sizes.

Despite these efforts, the Page Load Time (PLT) is still often unsatisfactorily high due to various reasons. One important issue among them is, when loading a web page, the network transfer and local computation (HTML parsing, script evaluation, page rendering) are interleaved, due to the complex dependencies among web objects (*i.e.,* resources). The key reason is, at the beginning of loading a page, a browser has no knowledge of what resources it will need. The page loading process therefore becomes a procedure of iteratively discovering, fetching, and consuming resources by expanding an initially unknown dependency graph.

[*]All authors made equal contribution.

AllThingsCellular'15, August 17-21, 2015, London, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3538-6/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785971.2785972

Such an interleaved resource exploration pattern degrades web performance and resource efficiency at various layers in cellular networks. At the radio layer, fetching data intermittently makes a device continuously occupy the high-power radio state and thus increases its radio energy consumption [17]. At the network layer, downloading resources in multiple round-trips increases the page load time, in particular in cellular networks with moderate to high RTT [19]. At the transport layer, a long idle gap may reset the congestion window and cause potential bandwidth under-utilization. At the application layer, fetching resources may block other activities (*e.g.,* synchronous JavaScript blocks HTML parsing [20]) and lengthens the critical path in the dependency graph. Quite a few work (*e.g.,* WebProphet [16], WProf [20], and Klotski [12]) has been done towards profiling and optimizing the dependency.

A different approach is to *break the dependency between network transfers and local computation* by obtaining all critical resources at the beginning of loading a page. In this way, most aforementioned inefficiencies at various layers will be eliminated or significantly mitigated. To realize this, two approaches have been proposed: Server Push and Server Hints. They allow a server to preemptively push the resource data, or to provide resource hints (*i.e.,* a list of resource URLs) to facilitate fetching. However, both approaches have limitations such as unnecessary push and additional latency, which are in particular undesired in cellular networks with metered links and high latency, as to be described in §2.

In this paper, we propose a cellular-friendly server push framework called MetaPush, which borrows ideas from Server Push and Server Hints, while overcoming their limitations. The key idea is to *explicitly provide a client with cacheable hints to facilitate early resource fetching*. Specifically, when loading a page, a server pushes hints (called *meta files*), which contain its own and its subpages' resource lists, to a client. The meta files are then cached by the client. Later, when the client is requesting a subpage with a cached meta file, it can then piggyback resource fetching requests with the page request. In this way, the whole page with its resources can be ideally downloaded in one round-trip with little extra bandwidth overhead due to small sizes of meta files. We present the design of MetaPush in §3.

MetaPush can be easily integrated with HTTP/2, the recently standardized next-generation HTTP [3], by strategically leveraging its built-in Server Push feature. It can also be implemented in today's HTTP/1.1 where push can be realized by add-ons such as WebSocket [9]. MetaPush is incrementally deployable with small changes on client browsers. If deployed at proxies (*e.g.,* Google's mobile web proxy [11]), no change needs to be made at remote servers. Our preliminary results indicate MetaPush is ideal for cellular networks: it incurs negligible bandwidth overhead compared to the bandwidth consumption of loading the actual page (median 0.4% across 20 popular websites). Moreover, MetaPush can reduce the PLT by up to 45% and the radio energy consumption by up to 37% over real HSPA+ networks (§4).

2. BACKGROUND AND MOTIVATION

We first review two existing relevant proposals: Server Push and Server Hints. We describe their strengths and limitations, which motivate our MetaPush proposal.

2.1 Server Push

Server Push is a feature that allows a server to preemptively send (*i.e.,* push) resources to a client without requiring the client to request the resource. The underlying assumption is the client will need the resource very shortly (*e.g.,* after parsing an HTML page), thus pushing it in advance avoids the round-trip delay and reduces the PLT. A pushed resource is usually cacheable so it will be loaded from the local cache when the client needs it.

Both HTTP/2 [3] and SPDY [7] support Server Push. In HTTP/2, to push a resource, the server first sends a **`PUSH_PROMISE`** frame on an existing client-initiated stream. This frame has two purposes. First, it notifies the client that the server will create a new stream for pushing by including the stream ID. Second, it contains the header information of the resource to be pushed. After sending the **`PUSH_PROMISE`** frame, the server initiates the new stream over which the resource data is sent.

Server Push can effectively reduce the PLT. However, its key limitation is that it is often difficult for server to determine what needs to be pushed [21]. In particular, if the client already has the unexpired resource in its cache, then pushing it is wasteful. Pushing unnecessary contents is in particular undesired in cellular networks where customers are charged by the amount of transferred bytes.

2.2 Server Hints

Server Hints provides a way for a server to notify a client of a resource that will be needed before it is discovered by the client. Unlike Server Push, when using Server Hints, the server does not send the actual resource data, but only its URL as a "hint". Therefore, the client only revalidates resources that are expired and requests resources that are not in the local cache.

Server Hints is realized using the **`Link`** header to be standardized in HTML 5. The server injects hints into response header using one of the two ways: **`Link:<url>;rel=prefetch`** [5] or **`Link:<url>;rel=subresource`** [6] where **`url`** is the resource URL. The browser then collects (potentially multiple) embedded hints, and (pre)fetches[1] them.

For the two methods above, **`Link rel=prefetch`** prefetches resources only when the browser is idle. In other words, the prefetching is given a low priority, and only happens *after* all regular resources in the current page are fetched and parsed. Therefore, this method is suitable for prefetching resources of *future* pages. **`Link rel=subresource`** has different semantics where fetching the hints is given a high priority *i.e.,* it happens *before* any regular resource in the current page is fetched. This method is hence suitable for early fetching resources of the *current* page.

Both methods address the client-side caching issue in Server Push. However, they both have limitations. **`Link rel=prefetch`** is difficult to use because prefetching happens whenever the browser is idle, and predicting which page the user is to browse next is hard. Aggressively prefetching all resources in all subpages will waste traffic, whereas conservatively prefetching only common objects in all subpages limits the applicability of prefetching. **`Link rel=subresource`** only works for the current page since the hints are embedded in the response header of the current page that needs to be requested first. Thus early fetching incurs an additional round-trip, which can be up to several hundred milliseconds in cellular networks.

[1]Throughout this paper, we use "prefetch" to denote getting resources of future pages, and use "fetch" or "early fetch" to refer to obtaining resources in the current page. MetaPush performs early fetching instead of prefetching.

3. MetaPush DESIGN

MetaPush is a novel server push mechanism aiming at reducing the page load time that is the dominating factor affecting users' web browsing experience. Its design needs to address a key challenge of *minimizing PLT while avoiding unnecessary data transfers*. In other words, we need a unified scheme that provides benefits of both Server Push and Server Hints while overcoming their limitations. Other challenges include incremental deployment, low runtime overhead, and minimal changes to the existing client/server software.

MetaPush works over any type of networks, including, in particular, metered cellular networks with moderate to high latency and high radio energy consumption. MetaPush consists of two phases: *push* and *early fetch*. In the push phase, a server (or proxy[2]) pushes several *meta files*, which contain resource URLs (*i.e.,* hints) of the current and/or future pages that are to be requested by a client. In the early fetch phase, the client requests for some or all resources according to the meta file, and batches the requests with the request of the page. In this way, ideally the page together with all its resources can be downloaded in one round-trip. This decouples the network transfer and local computation, and breaks the complex "load-parse-load" dependencies among objects, leading to reduced PLT [21].

3.1 Contents of a Meta File

In MetaPush, each page has a unique meta file. As listed in Table 1, a meta file has three parts. It begins with a header consisting of its meta file ID (mID), its associated page URL, and the expiration time. mIDs have one-to-one mappings with their pages, and can be simply generated by hashing the page URL[3]. The meta file itself is cacheable, with its own expiration time potentially differing from that of its associated page. The header is followed by a list of resources that a client may need when fetching the page which the meta file belongs to. Each entry of the resource contains the resource URL, and optionally a weight and the resource's expiration time/eTag. The optional fields will be described later. URLs in the list are properly ordered to observe the resource dependency (for local computation). Thus, the client can just issue early fetching requests according to the order[4]. The third part is a list of subpage mIDs. Subpages are pages which users can navigate to from the current page. They can be identified statically (by identifying HTML links) or statistically (*e.g.,* by observing from the web server logs to capture dynamic navigations). Figure 1 exemplifies three pages with their meta files.

[2]MetaPush can work with either a web proxy or a web server. For conciseness, we only use "server" in the rest of the paper.

[3]To handle dynamic pages, this can be generalized by associating a meta file with a URL pattern instead of a unique URL. See §5.

[4]As the requests arrive at the server in a single batch, the server can thus prioritize the corresponding responses for optimizing PLT. In that case, the order of the URLs does not matter.

Table 1: Contents of a meta file. "[...]" is an optional field.

```
//Part 1: a small header
meta file ID (mID), page URL, meta file expiration time
//Part 2: list of resources belonging to this page
resource URL 1, [weight], [expiration time], [eTag]
...
resource URL n, [weight], [expiration time], [eTag]
//Part 3: list of mIDs of subpages
mID of subpage 1, ..., mID of subpage m
```

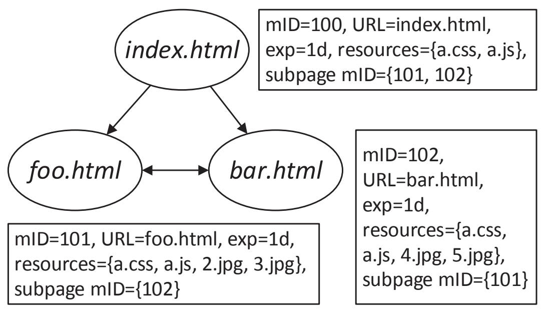
index.html
mID=100, URL=index.html, exp=1d, resources={a.css, a.js}, subpage mID={101, 102}
foo.html
bar.html
mID=102, URL=bar.html, exp=1d, resources={a.css, a.js, 4.jpg, 5.jpg}, subpage mID={101}
mID=101, URL=foo.html, exp=1d, resources={a.css, a.js, 2.jpg, 3.jpg}, subpage mID={102}

Figure 1: An example of web pages and their meta files. An arrow from page x to y indicates users can navigate from x directly to y.

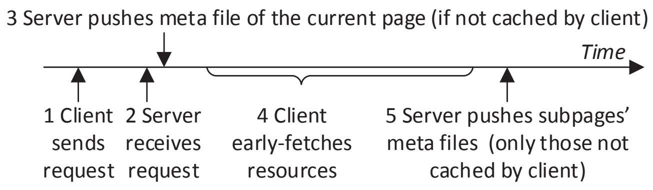
3 Server pushes meta file of the current page (if not cached by client)
Time
1 Client sends request
2 Server receives request
4 Client early-fetches resources
5 Server pushes subpages' meta files (only those not cached by client)

Figure 2: Timeline of two server pushes in MetaPush.

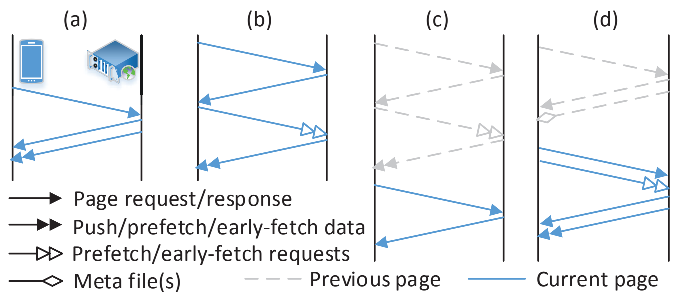

Figure 3: Illustrations of four push/prefetching/early fetching schemes: (a) Server Push, (b) Server Hints using `rel=subresource`, (c) Server Hints using `rel=prefetch`, and (d) MetaPush.

3.2 Pushing Meta Files

For each page, meta files are propagated to clients in (at most) two pushes, as shown in Figure 2. First, upon the reception of a page request, a server immediately pushes the page's meta file (Step 3). This helps a client early-fetch resources in the current page. Second, when becoming idle (*i.e.,* after delivering all essential resources of the current page), the server pushes subpages' meta files (Step 5). Recall in Table 1 that the meta file contains subpages' mIDs.

The traffic overhead of push is expected to be small given the small sizes of meta files (§4.1). In addition, two measures are employed to reduce the push size. First, a cap can be applied on the number of pushed meta files by either the client (using a request header) or the server. For example, based on subpages' popularity, we can push only the top-k meta files with little loss of the benefits from MetaPush. Second, the server does not need to push meta files that are already cached by the client. To realize this, MetaPush uses a lightweight cache hint and validation mechanism to inform the server of the set of cached meta files. Specifically, the client embeds a list into the page request. If the meta file of the current page does not exist, the list is empty. In this case the server will do a full push of meta files of the current page and all its subpages unless a cap is set (Step 3 and 5 in Figure 2). Otherwise, the list contains (mID, expiration time) pairs of the current page and subpages whose meta files are cached. Thus the server will (1) validate the meta files in the list and push updated versions if they are out-of-date, and (2) push meta files that belong to subpages but are not in the list. As a result, the client will have up-to-date meta files of the current page and its subpages.

We expect the overhead of the above mechanism to be small. Assuming a 4-byte mID and 6-byte expiration time, even validating a list of 100 meta files consumes only 1KB data in the request header, which can further be compressed in HTTP/2. Note the length of the list is bounded by the number of a page's subpages.

Example. In Figure 1, assuming a client is requesting the landing page `index.html`. Consider two scenarios. (1) If the client has a cached copy of meta file 100 and 101, it will let the server know that by embedding (100, exptime) and (101, exptime) in the request. The server will validate them, and then push the meta file 102 after the page is loaded because 102 is a subpage of 100 and is not currently cached. If the meta file 100 is fresh (the client knows that beforehand), loading `index.html` takes 1 round-trip in the ideal case. (2) If the client has an empty cache, the server will push the meta file 100 immediately when the request is received so the client can fetch `a.css` and `a.js`. After the page is loaded, the server pushes 101 and 102. The whole page load takes at least 2 round-trips, but the additional round-trip will be avoided for subsequent loadings of `foo.html` and `bar.html`.

3.3 Using Meta Files

Meta files can be leveraged by a client in several ways.

- A meta file provides a manifest for early-fetching resources at the beginning of loading a page.
- The client can use the *weight* field (Table 1) to better tradeoff between traffic volume and latency, by performing *selective* fetching. A weight is a number between 0 and 1, denoting the probability that the resource will be needed by the client. For resources that are dynamically requested (*e.g.,* by JavaScript), their weights can be less than 1. To reduce bandwidth consumption, the client can take a conservative approach by only fetching resources with large weights.
- The client can leverage the optional *expiration time* and *eTag* fields to conduct local cache validation for expired resources. In this way, network cache validation (*e.g.,* using `If-Modified-Since`) can be avoided to further reduce PLT.

3.4 Comparison with Server Push and Hints

We illustrate the comparison in Figure 3.

Compared to Server Push, MetaPush gives more flexibility to the client, which has much better knowledge of what needs to be fetched than the server does. This prevents pushing unnecessary data, leading to reduced network traffic.

Table 2: Statistics of meta file bundles.

Website Name	# Sub-pages	# Total Res	Size of meta file bundle in KB		
			plain	gzip	LZMA
stackoverflow.com	456	5,328	258 (0.6)	17 (0.0)	15 (0.0)
nytimes.com	213	8,290	732 (3.4)	66 (0.3)	51 (0.2)
foxnews.com	171	4,198	342 (2.0)	27 (0.2)	23 (0.1)
imgur.com	161	3,508	121 (0.8)	10 (0.1)	8 (0.1)
dealsea.com	158	2,773	132 (0.8)	9 (0.1)	8 (0.0)
en.wikipedia.org	154	6,039	603 (3.9)	85 (0.6)	67 (0.4)
kayak.com	145	11,001	741 (5.1)	74 (0.5)	54 (0.4)
abcnews.go.com	138	7,968	493 (3.6)	43 (0.3)	30 (0.2)
cnn.com	135	29,041	2,535 (19)	175 (1.3)	98 (0.7)
adobe.com	124	1,054	82 (0.7)	4 (0.0)	4 (0.0)
espn.go.com	121	3,362	227 (1.9)	21 (0.2)	17 (0.1)
npr.org	116	3,163	322 (2.8)	52 (0.5)	39 (0.3)
target.com	91	2,130	151 (1.7)	8 (0.1)	6 (0.1)
ca.gov	76	1,785	82 (1.1)	2 (0.0)	2 (0.0)
att.com	72	2,074	139 (1.9)	10 (0.1)	8 (0.1)
umich.edu	52	233	20 (0.4)	1 (0.0)	1 (0.0)
weather.com	49	968	97 (2.0)	8 (0.2)	7 (0.1)
sigcomm.org	37	903	53 (1.5)	1 (0.0)	1 (0.0)
apple.com	28	631	39 (1.4)	4 (0.1)	4 (0.1)
skype.com	19	377	26 (1.4)	2 (0.1)	2 (0.1)

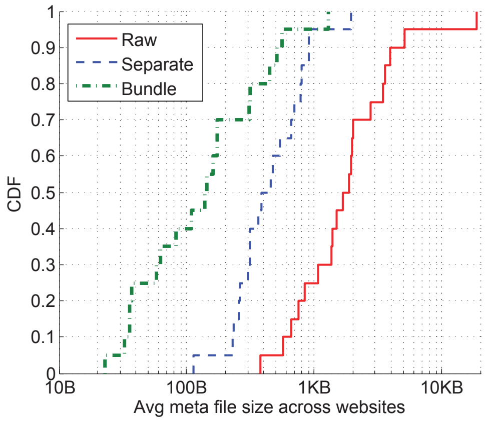

Figure 4: Average meta file size across all sites: uncompressed, gzip-ed individually, and gzip-ed into a bundle (showing average bundle contribution per page).

Our approach also overcomes key limitations of Server Hints. Compared to **`Link rel=prefetch`**, MetaPush employ more targeted early fetch. Instead of prefetching when the browser is idle, MetaPush delays that until requesting the actual page by explicitly associating resource lists (*i.e.,* meta files) with pages, in order to reduce unnecessary transfers. Compared to **`Link rel=subresource`**, MetaPush makes meta files cacheable and reusable, so the server does not need to attach the hints to *every* page's response. This eliminates additional round-trip for getting the hints, and makes hint delivery more efficient by bundling multiple pages' hints (to be described in §4.1). Finally, MetaPush combines the two Server Hints approaches into a unified framework, and provides additional benefits such as selective fetching and local cache validation.

It can be seen that meta files capture both *intra-page* metadata (*i.e.,* resources in a page) and *inter-page* metadata (*i.e.,* the connectivity among pages within a website). A lack of such information is the root cause of many web performance issues. Our approach thus facilitates early resource fetching by *making such information explicitly available to the client.*

4. PRELIMINARY RESULTS

We show two key results to demonstrate the feasibility of using MetaPush in cellular networks. (1) Meta files' sizes are small, especially when multiple meta files are bundled together (§4.1). (2) MetaPush can significantly reduce PLT and device radio energy consumption (§4.2).

4.1 Characterizing Meta Files

We conduct a measurement study of meta files using the following methodology. We pick 20 popular websites listed in Table 2. For each website, we used HTTrack Website Copier [2], an offline browsing tool, to download the landing page (*e.g.,* **`www.cnn.com`**) together with its all subpages by setting the maximum mirroring depth to 2. We then wrote a custom tool that parses each page, extracts URLs of all embedded resources, and constructs the page's meta file. Popular resources include images, JavaScript, and CSS files. Next, for each website, we created a meta file *bundle*, which is simply a file consisting of concatenation of all meta files, to take into account that multiple meta files are usually pushed in a batch in practice.

We noticed two limitations of the above approach. First, since we examine the pages statically, some resources dynamically requested by JavaScript might be missing in the generated meta files. Second, not all resources in the meta files will be fetched by the client since we are blindly looking for *any* resource address in a page when extracting them. Despite these limitations, we believe the results well approximate the true statistics about meta files in the wild.

The results are described in Table 2. Column 2 is the total number of subpages, and column 3 shows the total number of identified resources across all subpages and the landing page. Due to the diverse complexity across websites, the subpage and resource counts exhibit high variation. Column 4 to 6 measure the meta file bundle size in three encoding schemes: no compression, gzip compression, and LZMA compression [4]. The numbers in the parentheses correspond to the per-page contribution to the bundle (*i.e.,* its size divided by the number of pages).

We make two key observations. First, meta file bundles are highly compressible, because they are in plain text and more importantly, different pages of the same website share many common resources [22]. Due to this, as demonstrated in Figure 4, compressing multiple meta files into a bundle and pushing the whole bundle is preferred over compressing and pushing each meta file individually. In addition, as shown in Table 2, using advanced compression techniques (*e.g.,* LZMA) further reduces the size of meta file bundles.

Second, the meta file bundle sizes are small, with the 25-th, 50-th, and 75-th percentiles being 3.9KB, 8.3KB, and 32.6KB, respectively, for LZMA encoding across the 20 websites. The average per-page contribution (*i.e.,* bundle size / # pages) to the bundle is only 0.05KB, 0.1KB, and 0.2KB, respectively, for the 25-th, 50-th, and 75-th percentiles. Figure 5 plots the ratios of meta file bundle size to its landing page size (including its resources). The ratios are quite small for most websites (median: 0.4%). The only outlier is **`en.wikipedia.org`**, whose meta file bundle size (67KB in LZMA) is non-trivial compared to the small landing page size (410KB). In this case, the server can push only top k meta files as described in §3.2. Also recall that meta files are cacheable, and thus the bundle sizes shown in Table 2 and Figure 5 are in fact upper bounds.

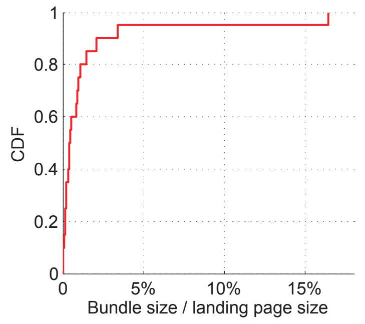

Figure 5: Ratios of bundled meta file size (LZMA encoding) to the landing page size (including resources in original encoding) across all sites.

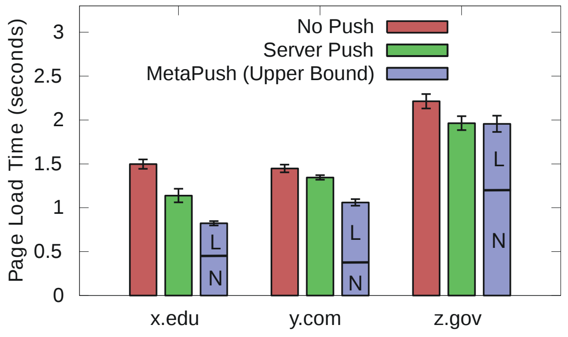

Figure 6: PLT for No Push, Server Push, and MetaPush. "N" and "L" denote time spent in network transfer and local computation, respectively. We report the upper bound PLT for MetaPush.

4.2 No Push, Server Push and MetaPush

We compare MetaPush with the default request-response-based page loading (No Push) and standard Server Push. (1) For No Push, we use off-the-shelf Chrome browser (version 41.0.2272.101) and Apache server 2.4.7 with `mod-spdy` version 0.9.4.1 in default settings. (2) For Server Push, we enable the push feature of the `mod-spdy` module. Since there is no standard about how to perform the push, we employ a simple policy of pushing all resources to the client. Studying other policies, which may consume less bandwidth but increase the PLT [21], is our future work. (3) We implemented an emulator of MetaPush, which consists of two parts: downloading all resources in a batch (using parallel async `XMLHttpRequest` [10]) and loading the whole web page offline from cache. We measure their consumed time using a Chrome extension and the Chrome DevTools, respectively, and estimate the overall PLT as the sum of the two components. It is important to note that, by doing so, the emulator only gives an *upper bound* (*i.e.,* worst case) of the PLT since in reality, the two components can potentially be performed in parallel, leading to a lower PLT. To make the experiments reproducible, we mirrored the landing pages of three websites from a university (`x.edu`), a retailer (`y.com`) and a state government (`z.gov`), by slightly modifying their source code to disable access to third-party sites and confine all traffic to our own Apache server. All three schemes use SPDY (v3.1) [7], which is the basis of HTTP/2.

Figure 6 plots the PLT for the three schemes over a commercial HSPA+ network, based on 10 runs of each scheme/website pair. We make three observations. First, MetaPush outperforms No Push by 45%, 27%, 11%, for the three sites, respectively, implying the potential performance benefits brought by decoupling the network transfer and local computation. Second, surprisingly, Server Push exhibits worse performance than MetaPush (ideally their PLTs should be similar). This is likely due to an implementation issue of SPDY's Server Push mechanism that needs further investigation[5]. Third, we show the local computation ("L") and the network transfer ("N") components of MetaPush's PLT. When the network transfer dominates the overall PLT (the `z.gov` case), the benefits of MetaPush diminishes. This is because `z.gov` has a large content size, and for all three schemes, the page loading is throttled by the limited HSPA+ bandwidth. We repeated the same experiment on an LTE network, and observed MetaPush brought more than 20% of PLT reduction compared to No Push.

Radio Energy Saving. We use an HSPA+ energy model based on [18] to study the radio energy saving brought by MetaPush. The radio power consumed by the cellular interface accounts for 1/3 to 1/2 of the overall handset power consumption [18]. Across the three sites, compared to the No Push case, MetaPush reduces the radio energy consumption by 25% to 37%. Note that we did not include the local computation energy that will likely be similar between the two schemes.

5. DISCUSSIONS

In this section, we discuss several remaining issues not addressed in §3 and §4.

Deployment considerations. For end-to-end deployment, our proposed MetaPush does require changes to both the client and the server. However, the MetaPush logic can be integrated into middleboxes to make it transparent to remote web servers. Such middleboxes have been widely deployed by, for example, cellular ISPs [14] and Google [11]. On the client side, MetaPush can be implemented as browser plugins that can even be developed by third parties. Note that MetaPush is *incrementally deployable* as the meta files will not be pushed unless a specific field containing the (potentially empty) cached mID list is found in the header request (§3.2). The key building blocks needed to realize MetaPush *i.e.,* push and batched fetching, have already been supported and standardized. MetaPush can be easily integrated with HTTP/2 where Server Push is a built-in feature. It can also be implemented on HTTP/1.1 where push can be realized by add-ons such as WebSocket [9].

Generating meta files. Meta files can be generated using a combination of several methods. Statically embedded resources can be identified by parsing the page offline. Server logs from many users can be leveraged to statistically capture resources that are dynamically requested by JavaScript [11], as well as their weights. Subpages can be identified using similar approaches.

Dynamic pages. In §3, we assume a one-to-one mapping between pages' URLs and their meta files. For dynamic pages, the URLs often contain parameters so ideally we need to associate a meta file with URLs showing the same pattern (*e.g.,* `foo.com/bar?para=*`) that contains the same or very similar resources. This can be addressed by using regular expressions (regex) instead of unique URL strings to match pages, and the mID can thus be generated by hashing the regex. We leave the design of the regex extraction algorithm (*e.g.,* based on server logs) as our future work.

[5]In our experiments, when Server Push is enabled, the Chrome Browser crashed several times.

HTTPS. MetaPush works well for HTTPS. However, as a well-known issue, handling HTTPS using a man-in-the-middle proxy breaks HTTPS' end-to-end security, unless the proxy is fully trusted [19].

6. RELATED WORK

We review existing work on understanding and improving mobile web performance besides Server Push and Server Hints.

Web object dependency is known to adversely affect the web performance. Prior systems such as WebProphet [16], WProf [20], and Klotski [12] either focus on the construction of the dependency graph, or use the graph for various purposes. WebProphet [16] is a system that captures the dependencies among web objects and automates the prediction of user-perceived web performance. It first constructs the dependency graph of a web page based on the observation that the delay of loading an object will propagate to all other objects that depend on it. It then simulates the web page load process and predicts the page load time for different optimizations. Wang *et al.* [20] built an in-browser profiler, called WProf, that can generate a dependency graph among the browser activities when loading a web page. Klotski [12] is a system that improves the quality of user experience for mobile web browsing by dynamically reprioritizing web resources. Instead of reducing page load time directly, the goal of Klotski is to deliver as many as possible high priority resources within the first 3-5 seconds when loading a web page. In contrast, MetaPush attempts to break the dependency between local computation and network transfers to address various inefficiencies (§1).

SPDY performance. Qian *et al.* investigated the prevalence of SPDY usage for the top 500 mobile websites [17]. Erman *et al.* [13] identified poor interaction between SPDY and TCP for cellular networks. Wang *et al.* [21] investigated how different factors, such as RTT, packet loss rate, and bandwidth, affect the performance of HTTP and SPDY. These studies provided valuable insight in various aspects of SPDY that is the base protocol for HTTP/2. Our proposal instead focuses a different aspect of strategically leveraging the server push feature.

Push. Wang *et al.* [21] demonstrated on emulated network that Server Push can reduce PLT. PARCEL [19] leverages a proxy to push web contents in one or several bundles to the mobile device, which reduces the PLT and the mobile device energy consumption. However, it suffers from the same limitation as Server Push: it is difficult for the proxy to know the clients' cache status. To address this issue, the proxy has to maintain per-client state, incurring overhead and complexity [19]. A recent proposal [15] attempts to address this by using *cache hints*: the client encodes its cached entries into several bloom filters and sends them to the server to facilitate Server Push. However, bloom filters may incur false positives as the cache becomes large. In contrast, MetaPush only pushes small meta files, and lets the client decide what to fetch.

Prefetching. Wang *et al.* [22] proposed a technique called *speculative loading* where the client predicts and prefetches subresources a web page will need. While MetaPush has a similar high-level concept, it makes a key difference that the server can simply provide the subresource list to the client. This leads to a simpler system, and eliminates the client-side learning overhead and prediction inaccuracies.

7. CONCLUDING REMARKS

We have described the MetaPush framework and demonstrated its feasibility in cellular networks. By explicitly exposing the intra-page and inter-page metadata to client browsers, our design trades off small bandwidth usage for significantly reduced PLT. For cellular networks, MetaPush can improve PLT of web browsing by up to 45% and save energy consumption on mobile devices by up to 37%. We are currently prototyping the MetaPush framework, with the goal of conducting field trials using cellular middleboxes and evaluating its effectiveness on real mobile web workloads.

Acknowledgments

We thank Lusheng Ji for initial discussions on the concept of MetaPush. We also thank anonymous reviewers for their valuable comments. This research was sponsored in part by Indiana University Faculty Research Support Program (FRSP) – Seed Funding.

8. REFERENCES

[1] Google modpagespeed. https://code.google.com/p/modpagespeed/.

[2] HTTrack Website Copier. https://www.httrack.com/.

[3] Hypertext Transfer Protocol version 2 (RFC 7540). https://tools.ietf.org/html/rfc7540.

[4] LZMA SDK. http://www.7-zip.org/sdk.html.

[5] Server Hint (prefetch). https://developer.mozilla.org/en-US/docs/Web/HTTP/Link_prefetching_FAQ.

[6] Server Hint (subresource). https://www.chromium.org/spdy/link-headers-and-server-hint/link-rel-subresource.

[7] SPDY Protocol – Draft 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[8] WebP: A new image format for the Web. https://developers.google.com/speed/webp/.

[9] WebSocket. https://www.websocket.org/.

[10] XMLHttpRequest. https://xhr.spec.whatwg.org/.

[11] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *NSDI*, 2015.

[12] 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.

[13] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier Mobile Web? In *CoNeXT*, 2013.

[14] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *SIGCOMM*, 2013.

[15] J. Khalid, S. Agarwal, A. Akella, and J. Padhye. Improving the performance of SPDY for mobile devices. In *HotMobile (Poster Session)*, 2015.

[16] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *NSDI*, 2010.

[17] F. Qian, S. Sen, and O. Spatscheck. Characterizing Resource Usage for Mobile Web Browsing. In *Mobisys*, 2014.

[18] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Mobisys*, 2011.

[19] A. Sivakumar, S. P. Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy Assisted BRowsing in Cellular networks for Energy and Latency reduction. In *CoNEXT*, 2014.

[20] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *NSDI*, 2013.

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

[22] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Fast Can Client-Only Solutions Go for Mobile Browser Speed. In *WWW*, 2012.