# How speedy is SPDY?

*Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall*
*University of Washington*

## Abstract

SPDY is increasingly being used as an enhancement to HTTP/1.1. To understand its impact on performance, we conduct a systematic study of Web page load time (PLT) under SPDY and compare it to HTTP. To identify the factors that affect PLT, we proceed from simple, synthetic pages to complete page loads based on the top 200 Alexa sites. We find that SPDY provides a significant improvement over HTTP when we ignore dependencies in the page load process and the effects of browser computation. Most SPDY benefits stem from the use of a single TCP connection, but the same feature is also detrimental under high packet loss. Unfortunately, the benefits can be easily overwhelmed by dependencies and computation, reducing the improvements with SPDY to 7% for our lower bandwidth and higher RTT scenarios. We also find that request prioritization is of little help, while server push has good potential; we present a push policy based on dependencies that gives comparable performance to mod_spdy while sending much less data.

## 1 Introduction

HTTP/1.1 has been used to deliver Web pages using multiple, persistent TCP connections for at least the past decade. Yet as the Web has evolved, it has been criticized for opening too many connections in some settings and too few connections in other settings, not providing sufficient control over the transfer of Web objects, and not supporting various types of compression.

To make the Web faster, Google proposed and deployed a new transport for HTTP messages, called SPDY, starting in 2009. SPDY adds a framing layer for multiplexing concurrent application-level transfers over a single TCP connection, support for prioritization and unsolicited push of Web objects, and a number of other features. SPDY is fast becoming one of the most important protocols for the Web; it is already deployed by many popular websites such as Google, Facebook, and Twitter, and supported by browsers including Chrome, Firefox, and IE 11. Further, IETF is standardizing a HTTP/2.0 proposal that is heavily based on SPDY [10].

Given the central role that SPDY is likely to play in the Web, it is important to understand how SPDY performs relative to HTTP. Unfortunately, the performance of SPDY is not well understood. There have been several studies, predominantly white papers, but the findings often conflict. Some studies show that SPDY improves performance [20, 14], while others show that it provides only a modest improvement [13, 19]. In our own study [25] of page load time (PLT) for the top 200 Web pages from Alexa [1], we found either SPDY or HTTP could provide better performance by a significant margin, with SPDY performing only slightly better than HTTP in the median case.

As we have looked more deeply into the performance of SPDY, we have come to appreciate why it is challenging to understand. Both SPDY and HTTP performance depend on many factors external to the protocols themselves, including network parameters, TCP settings, and Web page characteristics. Any of these factors can have a large impact on performance, and to understand their interplay it is necessary to sweep a large portion of the parameter space. A second challenge is that there is much variability in page load time (PLT). The variability comes not only from random events like network loss, but from browser computation (i.e., JavaScript evaluation and HTML parsing). A third challenge is that dependencies between network activities and browser computation can have a significant impact on PLT [25].

In this work, we present what we believe to be the most in-depth study of page load time under SPDY to date. To make it possible to reproduce experiments, we develop a tool called `Epload` that controls the variability by recording and replaying the process of a page load at fine granularity, complete with browser dependencies and deterministic computational delays; in addition we use a controlled network environment. The other key to our approach is to isolate the different factors that affect PLT with reproducible experiments that progress from simple but unrealistic transfers to full page loads. By looking at results across this progression, we can systematically isolate the impact of the contributing factors and identify when SPDY helps significantly and when it performs poorly compared to HTTP.

Our experiments progress as follows. We first compare SPDY and HTTP simply as a transport protocol (with no browser dependencies or computation) that transfers Web objects from both artificial and real pages (from the top 200 Alexa sites). We use a decision tree analysis to identify the situations in which SPDY outperforms HTTP and vice versa. We find that SPDY improves PLT significantly in a large number of scenarios that track the benefits of using a single TCP connection. Specifically, SPDY helps for small object sizes and under low loss rates by: batching several small objects in a TCP segment; reducing congestion-induced retransmis-

sions; and reducing the time when the TCP pipe is idle. Conversely, SPDY significantly hurts performance under high packet loss for large objects. This is because a set of TCP connections tends to perform better under high packet loss; it is necessary to tune TCP behavior to boost performance.

Next, we examine the complete Web page load process by incorporating dependencies and computational delays. With these factors, the benefits of SPDY are reduced, and can even be negated. This is because: i) there are fewer outstanding objects at a given time; ii) traffic is less bursty; and iii) the impact of the network is degraded by computation. Overall, we find SPDY benefits to be larger when there is less bandwidth and longer RTTs. For these cases SPDY reduces the PLT for 70–80% of Web pages, and for shorter, faster links it has little effect, but it can also increase PLT: the worst 20% of pages see an increase of at least 6% for long RTT networks.

In search of greater benefits, we explore SPDY mechanisms for prioritization and server push. Prioritization helps little because it is limited by load dependencies, but server push has the potential for significant improvements. How to obtain this benefit depends on the server push policy, which is a non-trivial issue because of caching. This leads us to develop a policy based on dependency levels that performs comparably to mod_spdy's policy [11] while pushing 80% less data.

Our contributions are as follows:

- A systematic measurement study using synthetic pages and real pages from 200 popular sites that identifies the combinations of factors for which SPDY improves (and sometimes reduces) PLT compared to HTTP.
- A page load tool, `Epload`, that emulates the detailed page load process of a target page, including its dependencies, while eliminating variability due to browser computation. With a controlled network environment, `Epload` enables reproducible but authentic page load experiments for the first time.
- A SPDY server push policy based on dependency information that provides comparable benefits to mod_spdy while sending much less data over the network.

In the rest of this paper, we first review SPDY background (§2) and then briefly describe our challenge and approach (§3). Next, we extensively study TCP's impact on SPDY (§4) and extend to Web page's impact on SPDY (§5). We discuss in §6, review related work in §7, and conclude in §8.

# 2 Background

In this section, we review issues with HTTP performance and describe how the new SPDY protocol addresses them.

## 2.1 Limitations of HTTP/1.1

When HTTP/1.1, or simply HTTP, was designed in the late 1990's, Web applications were fairly simple and rudimentary. Since then, Web pages have become more complex and dynamic, making it difficult for HTTP to meet the increasingly demanding user experience. Below, we identify some of the limitations of HTTP:

*i) Browsers open too many TCP connections to load a page.* HTTP improves performance by using parallel TCP connections. But if the number of connections is too large, the aggregate flow may cause network congestion, high packet loss, and reduced performance [9]. Further, services often deliver Web objects from multiple domains, which results in even more TCP connections and the possibility of high packet loss.

*ii) Web transfers are strictly initiated from the client.* Consider the loading of embedded objects. Theoretically, the server can send embedded objects along with the parent object when it receives a request for the parent object. In HTTP, because an object can be sent only in response to a client request, the server has to wait for an explicit request which is sent only after the client has received and processed the parent page.

*iii) A TCP segment cannot carry more than one HTTP request or response.* HTTP, TCP and other headers could account for a significant portion of a packet when HTTP requests or responses are small. So if there are a large number of small embedded objects in a page, the overhead associated with these headers is substantial.

## 2.2 SPDY

SPDY addresses several of the issues described above. We now review the key ideas in SPDY's design and implementation and its deployment status.

**Design:** There are four key SPDY features.

*i) Single TCP connection.* SPDY opens a single TCP connection to a domain and multiplexes multiple HTTP requests and responses (a.k.a., SPDY streams) over the connection. The multiplexing here is similar to HTTP/1.1 pipelining but is finer-grained. A single connection also helps reduce SSL overhead. Besides client-side benefits, using a single connection helps reduce the number of TCP connections opened at servers.

*ii) Request prioritization.* Some Web objects, such as JavaScript code modules, are more important than others and thus should be loaded earlier. SPDY allows the client to specify a priority level for each object, which is then used by the server in scheduling the transfer of the object.

*iii) Server push.* SPDY allows the server to push embedded objects before the client requests for them. This improves latency but could also increase transmitted data if the objects are already cached at the client.

*iv) Header compression.* SPDY supports HTTP

header compression since experiments suggest that HTTP headers for a single session contain duplicate copies of the same information (e.g., `User-Agent`).

**Implementation:** SPDY is implemented by adding a framing layer to the network stack between HTTP and the transport layer. Unlike HTTP, SPDY splits HTTP headers and data payloads into two kinds of frames. `SYN_STREAM` frames carry request headers and `SYN_REPLY` frames carry response headers. When a header exceeds the frame size, one or more `HEADERS` frames will follow. HTTP data payloads are sliced into `DATA` frames. There is no standardized value for the frame size, and we find that mod_spdy caps frame size to 4KB [11]. Because frame size is the granularity of multiplexing, too large a frame decreases the ability to multiplex while too small a frame increases overhead. SPDY frames are encapsulated in one or more consecutive TCP segments. A TCP segment can carry multiple SPDY frames, making it possible to batch up small HTTP requests and responses.

**Deployment:** SPDY is deployed over SSL and TCP. On the client side, SPDY is enabled in Chrome, Firefox, and IE 11. On the server side, popular websites such as Google, Facebook, and Twitter have deployed SPDY. Another popular use of SPDY is between a proxy and a client, such as the Amazon Silk browser [16] and Android Chrome Beta [2]. SPDY version 3 is the most recent specification and is widely deployed [21].

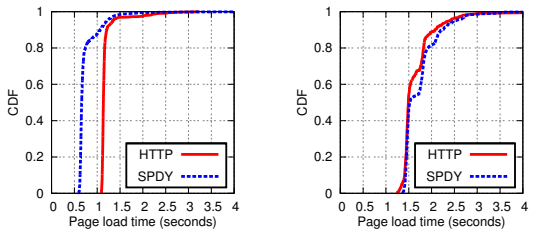
## 3 Pinning SPDY down

We would like to experimentally evaluate how SPDY performs relative to HTTP because SPDY is likely to play a key role in the Web. But, understanding SPDY performance is hard. Below, we identify three challenges in studying the performance of SPDY and then provide an overview of our approach.

### 3.1 Challenges

We identify the challenges on the basis of previous studies and our own initial experimentation. As a first step, we extensively load two Web pages for a thousand times using a measurement node at the University of Washington. One page displays fifty world flags [12], which is advertised by mod_spdy [11] to demonstrate the performance benefits of SPDY, and the other is the Twitter home page. The results are depicted in Figure 1.

First, we observe that SPDY helps the flag page but not the Twitter page, and it is not immediately apparent as to why that is the case. Further experimentation in emulated settings also revealed that both the magnitude and the direction of the performance differences vary significantly with network conditions. Taken together, this indicates that SPDY's performance depends on many factors such as Web page characteristics, network parameters, and TCP settings, and that measurement studies will




(a) The flag page (b) Twitter home page

Figure 1: Distributions of PLTs of SPDY and HTTP. Performed a thousand runs for each curve without caching.

likely yield different, even conflicting, results, if they use different experimental settings. Therefore, a comprehensive sweep of the parameter space is necessary to evaluate under what conditions SPDY helps, what kinds of Web pages benefit most from SPDY, and what parameters best support SPDY.

Second, we observed in our experiments that the measured page load times have high variances, and this often overwhelms the differences between SPDY and HTTP. For example, in Figure 1(b), the variance of the PLT for the Twitter page is 0.5 second but the PLT difference between HTTP and SPDY is only 0.02 second. We observe high variance even when we load the two pages in a fully controlled network. This indicates that the variability likely stems from browser computation (i.e., JavaScript evaluation and HTML parsing). Controlling this variability is key to reproducing experiments so as to obtain meaningful comparisons.

Third, prior work has shown that the dependencies between network operations and computation has a significant impact on PLT [25]. Interestingly, page dependencies also influence the scheduling of network traffic and affects how much SPDY helps or hurts performance (§4 and §5). Thus, on one hand, ignoring browser computations can reduce PLT variability, but on the other hand, dependencies need to be preserved in order to obtain accurate measurements under realistic offered loads.

### 3.2 Approach

Our approach is to separate the various factors that affect SPDY and study them in isolation. This allows us to control and identify the extent to which these factors affect SPDY.

First, we extensively sweep the parameter space of all the factors that affect SPDY including RTT, bandwidth, loss rate, TCP initial window, number of objects on a page, and object sizes. We initially *ignore* page load dependencies and computation in order to simplify our analysis. This systematic study allows us to identify when SPDY helps or hurts and characterize the importance of the contributing factors. Based on further analysis of why SPDY sometimes hurts, we propose some

simple modifications to TCP.

Second, before we perform experiments *with* page load dependencies, we address the variability caused by computation. We develop a tool called `Epload` that emulates the process of a page load. Instead of performing real browser computation, `Epload` records the process of a sample page load, identifies when computations happen, and replays the page load by introducing the appropriate delays associated with the recorded computations. After emulating a computation activity, `Epload` performs real network requests to dependent Web objects. This allows us to control the variability of computation while also modeling page load dependencies. In contrast to the methodology that statistically reduces variability by obtaining a large amount of data (usually from production), our methodology mitigates the root cause of variability and thus largely reduces the amount of required experiments.

Third, we study the effects of dependencies and computation by performing page loads with `Epload`. We are then able to identify how much dependencies and computation affect SPDY, and to identify the relative importance of other contributing factors. To mitigate the negative impact of dependencies and computation, we explore the use of prioritization and server push that enable the client and the server to coordinate the transfers. Here, we are able to evaluate the extent to which these mechanisms can improve performance when used appropriately.

# 4 TCP and SPDY

In this section, we extensively study the performance of SPDY as a transfer protocol on both synthetic and real pages by ignoring page load dependencies and computation. This allows us to measure SPDY performance without other confounding factors such as browser computation and page load dependencies. Here, SPDY is only different from HTTP in the use of a single TCP connection, header compression, and a framing layer.

## 4.1 Experimental setup

We conduct the experiments by setting up a client and a server that can communicate over both HTTP and SPDY. Both the server and the client are connected to the campus LAN at the University of Washington. We use Dummynet [6] to vary network parameters. Below details the experimental setup.

**Server:** Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory. It runs Ubuntu 12.04 with Linux kernel 3.7.5 using the default TCP variant Cubic. We use a TCP initial window size of ten as the default setting, as suggested by SPDY best practices [18]. HTTP and SPDY are enabled on Apache 2.2.2 with the SPDY module, mod_spdy 0.9.3.3-386, installed. We use SPDY 3 without SSL which allows us to decode the SPDY frames in TCP payloads. To control the exact size of Web objects, we turn off gzip encoding.

| Categ | Factor | Range | High |
|---|---|---|---|
| Net | rtt | 20ms, 100ms, 200ms | $\geq$100ms |
| | bw | 1Mbps, 10Mbps | $\geq$10Mbps |
| | pkt loss | 0, 0.005, 0.01, 0.02 | $\geq$ 0.01 |
| TCP | iw | 3, 10, 21, 32 | $\geq$ 21 |
| Page | obj size | 100B, 1K, 10K, 100K, 1M | $\geq$ 1K |
| | # of obj | 2, 8, 16, 32, 64, 128, 512 | $\geq$ 64 |

Table 1: Contributing factors to SPDY performance. We define a threshold for each factor, so that we can classify a setting as being high or low in our analysis.

**Client:** Because we issue requests at the granularity of Web objects and not pages, we do not work with browsers, and instead develop our own SPDY client by following the SPDY/3 specification [21]. Unlike other wget-like SPDY clients such as `spdylay` [22] that open a TCP connection per request, our SPDY client allows us to reuse TCP connections. Similarly, we also develop an HTTP client for comparison. We set the maximum number of parallel TCP connections for HTTP to six, as used by all major browsers. As the receive window is autotuned, it is not a bottleneck in our experiments.

**Web pages:** To experiment with synthetic pages, we create objects with pre-specified sizes and numbers. To experiment with real pages, we download the home pages of the Alexa top 200 websites to our own server. To avoid the negative impact of domain sharding on SPDY [18], we serve all embedded objects from the same server including those that are dynamically generated by JavaScript.

We run the experiments presented in the entire paper from June to September, 2013. We repeat our experiments five times and present the median to exclude the effects of random loss. We collect network traces at both the client and the server. We define page load time (PLT) as the elapsed time between when the first object is requested and when the last object is received. Because we do not experiment within a browser, we do not use the W3C `load` event [24].

## 4.2 Experimenting with synthetic pages

In experimenting with synthetic pages, we consider a broad range of parameter settings for the various factors that affect performance. Table 1 summarizes the parameter space used in our experiments. The RTT values include 20ms (intra-coast), 100ms (inter-coast), and 200ms (3G link or cross-continent). The bandwidths emulate a broadband link with 10Mbps [4] and a 3G link with 1Mbps [3]. We inject random packet loss rates from zero to 2% since studies suggest that Google servers experience a loss rate between 1% and 2% [5]. At the server,

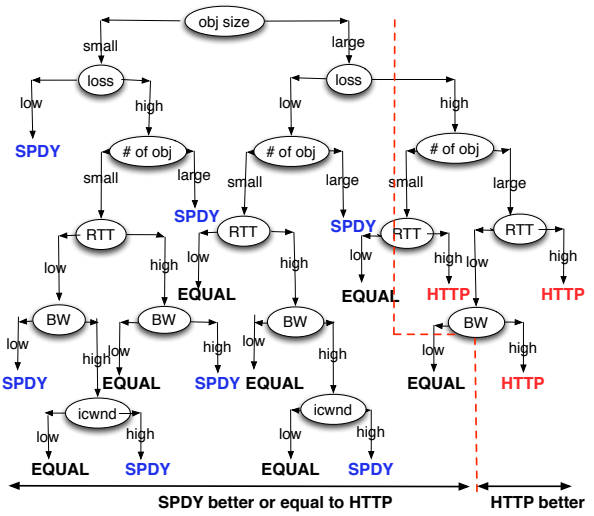



Figure 2: The decision tree that tells when SPDY or HTTP helps. A leaf pointing to SPDY (HTTP) means SPDY (HTTP) helps; a leaf pointing to EQUAL means SPDY and HTTP are comparable. Table 1 shows how we define a factor being high or low.

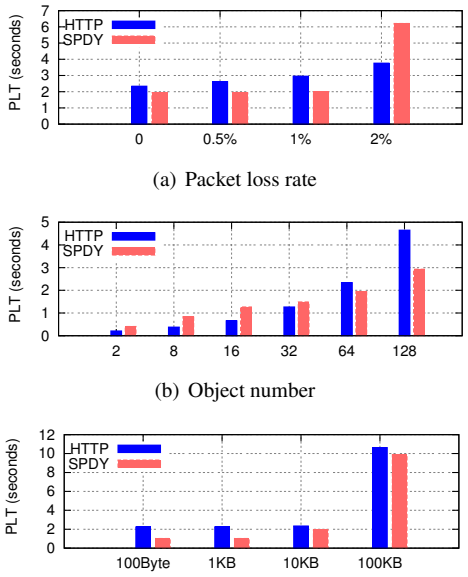
we vary TCP initial window size from 3 (used by earlier Linux kernel versions) to 32 (used by Google servers). We also consider a wide range of Web object sizes (100B to 1M) and object numbers (2 to 512). For simplicity, we choose one value for each factor which means that there is no cross traffic.

When we sweep this large parameter space, we find that SPDY improves performance under certain conditions, but degrades performance under other conditions.

#### 4.2.1 When does SPDY help or hurt

There have been many hypotheses as to whether SPDY helps or hurts based on analytical inference about parallel versus single TCP connections. For example, one hypothesis is that SPDY hurts because a single TCP connection increases congestion window slower than multiple connections; another hypothesis is that SPDY helps stragglers because HTTP has to balance its communications across parallel TCP. However, it is unclear how much hypotheses contribute to SPDY performance. Here, we sort out the most important findings, meaning that hypotheses that are shown here contribute more to SPDY performance than those that are not shown.

**Methodology:** To understand the conditions under which SPDY helps or hurts, we build a predictive model based on decision tree analysis. In the analysis, each configuration is a combination of values for all factors listed in Table 1. For each configuration, we add an additional variable $s$, which is the PLT of SPDY divided by that of HTTP. We run the decision tree to predict the configurations under which SPDY outperforms HTTP ($s < 0.9$) and under which HTTP outperforms SPDY ($s > 1.1$). The decision tree analysis generates the likelihood that a configuration works better under SPDY (or HTTP). If this likelihood is over 0.75, we mark the branch as SPDY (or HTTP); otherwise, we say that SPDY and HTTP perform equally.




(a) Packet loss rate

(b) Object number

(c) Object size

Figure 3: Performance trends for three factors with a default setting: rtt=200ms, bw=10Mbps, loss=0, iw=10, obj_size=10K, obj_number=64.

We obtain the decision tree in Figure 2 as follows. First, we produce a decision tree based on all the factors. To populate the branches, we also generate supplemental decision trees based on subsets of factors. Each supplemental decision tree has a prediction accuracy of 84% or higher. Last, we merge the branches from supplemental decision trees into the original decision tree.

**Results:** The decision tree shows that SPDY hurts when packet loss is high. However, SPDY helps under a number of conditions, for example, when there are:

- Many small objects, or small objects under low loss.
- Many large objects under low loss.
- Few objects under good network conditions and a large TCP initial window.

The decision tree also depicts the relative importance of contributing factors. Intuitively, factors close to the root of the decision tree affect SPDY performance more than those near the leaves. This is because the decision tree places the important factors near the root to reduce the number of branches. We find that object size and loss rate are the most important factors in predicting SPDY performance. However, RTT, bandwidth, and TCP initial window play a less important role.

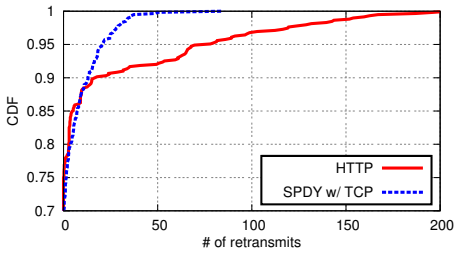



Figure 4: SPDY reduces the number of retransmissions.

**How much SPDY helps or hurts:** We present three trending graphs in Figure 3. Figure 3(a) shows that HTTP outperforms SPDY by half when loss rate increases to 2%, Figure 3(b) shows the trend that SPDY performs better as the number of objects increases, and Figure 3(c) shows the trend that SPDY performs worse as the object size increases. We publish the results, trends, and network traces at `http://wprof.cs.washington.edu/spdy/`.

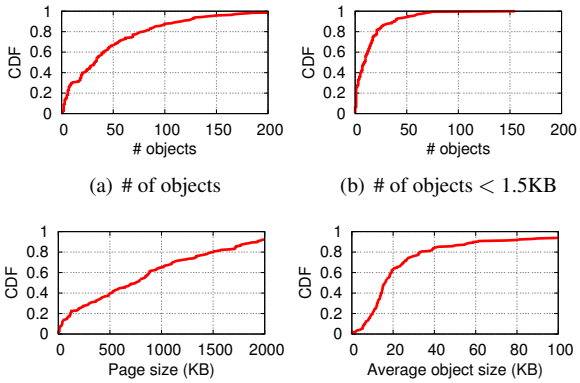
#### 4.2.2 Why does SPDY help or hurt

While the decision tree informs the conditions under which SPDY helps or hurts, it does not explain why. To this end, we analyze the network traces we collected to explain SPDY performance. We discuss below our findings.

*SPDY helps on small objects.* Our traces suggest that TCP implements congestion control by counting outstanding packets not bytes. Thus, sending a few small objects with HTTP will promptly use up the congestion window, though outstanding bytes are far below the window limit. In contrast, SPDY batches small objects and thus eliminates this problem. This explains why the flag page [12], which mod_spdy advertised, benefits from SPDY.

*SPDY benefits from having a single connection.* We find several reasons as to why SPDY benefits from a single TCP connection. First, a single connection results in fewer retransmissions. Figure 4 shows the retransmissions in SPDY and HTTP across all configurations except those with zero injected loss. SPDY helps because packet loss occurs more often when concurrent TCP connections are competing with each other. There are additional explanations for why SPDY benefits from using a single connection. In our previous study [25], our experiments showed that SPDY significantly reduced the contribution of the TCP connection setup time to the *critical path* of a page download. Further, our experiments in §5 will show that a single pipe reduces the amount of time the pipe is idle due to delayed client requests.

*SPDY degrades under high loss due to the use of a single pipe.* We discussed above that a single TCP connection helps under several conditions. However, a single connection hurts under high packet loss because it aggressively reduces the congestion window compared to HTTP which reduces the congestion window on only one of its parallel connections.




(a) # of objects (b) # of objects < 1.5KB

(c) Page size (KB) (d) Mean object size (KB)

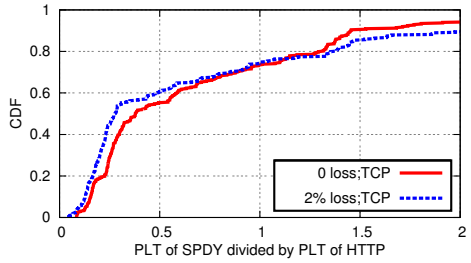
Figure 5: Characteristics of top 200 Alexa Web pages.

### 4.3 Experimenting with real pages

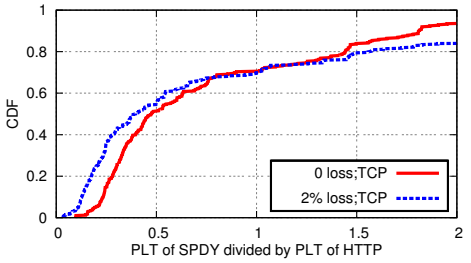
In this section, we study the effects of varying object sizes and number of objects based on the distributions observed in real Web pages. We continue to vary other factors such as network conditions and TCP settings based on the parameter space described in Table 1. Due to space limit, we only show results under a 10Mbps bandwidth.

First, we examine the page characteristics of real pages because they can explain why SPDY helps or hurts when we relate them to the decision tree. Figure 5 shows the characteristics of the top 200 Alexa Web pages [1]. The median number of objects is 30 and the median page size is 750KB. We find high variability in the size of objects within a page. The standard deviation of the object size within a page is 31KB (median), even more than the average object size 17KB (median).

Figure 6 shows PLT of SPDY divided by that of HTTP across the 200 Web pages. It suggests that SPDY helps on 70% of the pages consistently across network conditions. Interestingly, SPDY shows a 2x speedup over half of the pages, likely due to the following reasons. First, SPDY almost eliminates retransmissions (as indicated in Figure 7). Compared to a similar analysis for artificial pages (see Figure 4), SPDY's retransmission rate is even lower. Second, we find in Figure 5(b) that 80% of the pages have small objects, and that half of the pages have more than ten small objects. Since SPDY helps with small objects (based on the decision tree analysis), it is not surprising that SPDY has lower PLT for this set of experiments. In addition, we hypothesize that SPDY could help with stragglers since it multiplexes all objects on to a single connection and thus reduces the dynamics of congestion windows. To check this hypothesis, we ran a set of experiments with overall page size and the




(a) rtt=20ms, bw=10Mbps




(b) rtt=200ms, bw=10Mbps

Figure 6: SPDY performance across 200 pages with object sizes and numbers of objects drawn from real pages. SPDY helps more under a 1Mbps bandwidth.

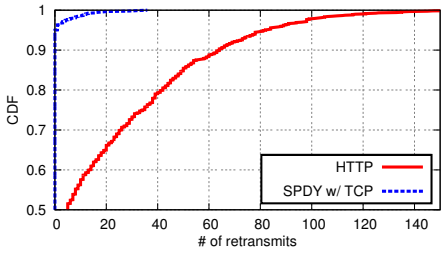



Figure 7: SPDY helps reduce retransmissions.

number of objects drawn from the real pages, but with equal object sizes embedded inside the pages. When we perform this experiment, HTTP's performance improves only marginally indicating that there is very little straggler effect.

### 4.4 TCP modifications

Previously, we found that SPDY hurts mainly under high packet loss because a single TCP connection reduces the congestion window more aggressively than HTTP's parallel connections. Here, we demonstrate that the negative impact can be mitigated by simple TCP modifications.

Our modification (a.k.a., TCP+) mimics behaviors of concurrent connections with a single connection. Let the number of parallel TCP connections be $n$. First, we propose to multiply the initial window by $n$ to reduce the effect of slow start. Second, we suggest scaling the receive window by $n$ to ensure that the SPDY connection has the same amount of receive buffer as HTTP's parallel connections. Third, when packet loss occurs, the congestion window (*cwnd*) backs off with a rate $\beta' = 1-(1-\beta)/n$ where $\beta$ is the original backoff rate. In practice, the number of concurrent connections changes over time. Because we are unable to pass this value to the Linux kernel in real time, we assume that HTTP uses six connections and set $n = 6$. We use six here because it is found optimal and used by major browsers [17].

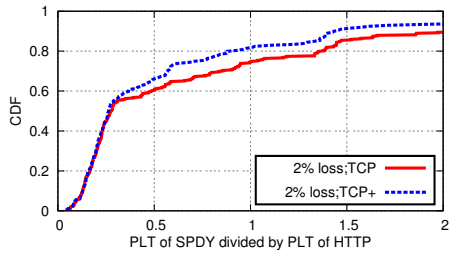



Figure 8: TCP+ helps SPDY across the 200 pages. RTT=20ms, BW=10Mbps. Results on other network settings are similar.

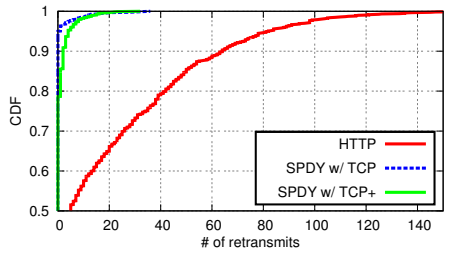



Figure 9: With TCP+, SPDY still produces few retransmissions.

We perform the same set of SPDY experiments with both synthetic and real pages using TCP+. Figure 8 shows that SPDY performs better with TCP+, and the decision tree analysis for TCP+ suggests that loss rate is no longer a key factor that determines SPDY performance.

To evaluate the potential side effects of TCP+, we look at the number of retransmissions produced by TCP+. Figure 9 shows that SPDY still produces much fewer retransmissions with TCP+ than with HTTP, meaning that TCP+ does not abuse the congestion window under the conditions that we experimented with. Here, we aim to demonstrate that SPDY's negative impact under high random loss can be mitigated by tuning the congestion window. Because the loss patterns in real networks are likely more complex, a solution for real networks requires further consideration and extensive evaluations and is out of the scope of this paper.

## 5 Web pages and SPDY

This section examines how SPDY performs for real Web pages. Real page loads incur dependencies and computation that may affect SPDY's performance. To incorporate dependencies and computation while controlling variability, we develop a page load emulator `Epload`

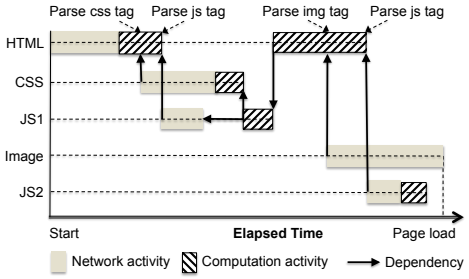


Figure 10: A dependency graph obtained from WProf.

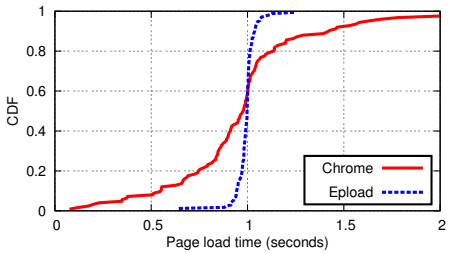


Figure 11: Page loads using Chrome v.s. Epload.

that hides the complexity and variations in browser computation while performing authentic network requests (§5.1). We use `Epload` to identify the effect of page load dependencies and computation on SPDY's performance (§5.2). We further study SPDY's potential by examining prioritization and server push (§5.3).

### 5.1 Epload: emulating page loads

Web objects in a page are usually not loaded at the same time, because loading an object can depend on loading or evaluating other objects. Therefore, not only network conditions, but also page load dependencies and browser computation, affect page load times. To study how much SPDY helps the overall page load time, we need to evaluate SPDY's performance by preserving dependencies and computation of real page loads.

Dependencies and computation are naturally preserved by loading pages in real browsers. However, this procedure incurs high variances in page load times that stem from both network conditions and browser computation. We have conducted controlled experiments to control the variability of network, and here introduce the `Epload` emulator to control the variability of computation.

**Design:** The key idea of `Epload` is to decouple network operations and computation in page loads. This allows `Epload` to simplify computation while scheduling network requests at the appropriate points during the page load.

`Epload` records the process of a page load by capturing the dependency graph using our previous work, WProf [25]. WProf captures the dependency and timing information of a page load. Figure 10 shows an example of a dependency graph obtained from WProf where activities depend on each other. This Web page embeds a CSS, a JavaScript, an image, and another JavaScript. A bar represents an activity (i.e., loading objects, evaluating CSS and JavaScript, parsing HTML) while an arrow represents that one activity depends on another. For example, evaluating JS1 depends on both loading JS1 and evaluating CSS. Therefore, evaluating JS1 can only start after the other two activities complete. There are other dependencies such as layout and painting. Because they do not occur deterministically and significantly, we exclude them here.

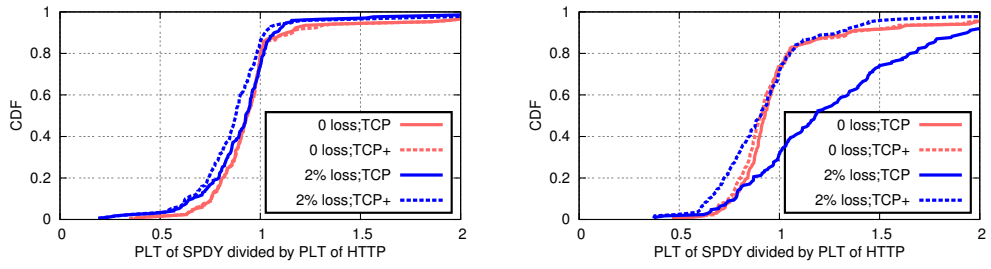
Using the recorded dependency graph, `Epload` replays the page load process as follows. First, `Epload` starts the activity that loads the root HTML. When the activity is finished, `Epload` checks whether it should trigger a dependent activity based on whether all activities that the dependent activity depends on are finished. For example in Figure 10, the dependent activity is parsing the HTML, and it should be triggered. Next, it starts the activity that parses the HTML. Instead of performing HTML parsing, it waits for the same amount of time that parsing takes (based on the recorded information) and checks dependent activities upon completion. This proceeds until all activities are finished. The actual replay process is more complex because a dependent activity can start before an activity is fully completed. For example, parsing an HTML starts after the first chunk of the HTTP response is received; and loading the CSS starts after the first chunk of HTML is fully parsed. `Epload` models all of these aspects of a page load.

**Implementation:** `Epload` recorder is implemented based on WProf to generate a dependency graph that specifies activities and their dependencies. `Epload` records the computational delays while performing the page load in the browser, whereas the network delays are realized independently for each replay run. We implement `Epload` replayer using node.js. The output from `Epload` replayer is a series of throttled HTTP or SPDY requests to perform a page load. The `Epload` code is available at `http://wprof.cs.washington.edu/spdy/`.

**Evaluation:** We validate that `Epload` controls the variability of computation. We compare the differences of two runs across 200 pages loaded by `Epload` and by Chrome. The network is tuned to a 20ms RTT, a 10Mbps bandwidth, and zero loss. Figure 11 shows that `Epload` produces at most 5% differences for over 80% of pages which is a 90% reduction compared to Chrome.

### 5.2 Effects of dependencies and computation

We use `Epload` to measure the impact of dependencies and computation. We set up experiments as follows. The `Epload` recorder uses a WProf-instrumented Chrome to




(a) rtt=20ms, bw=1Mbps

(b) rtt=200ms, bw=1Mbps

(c) rtt=20ms, bw=10Mbps

(d) rtt=200ms, bw=10Mbps

Figure 12: SPDY performance using emulated page loads. Compared to Figure 6, it suggests that dependencies and computation reduce the impact of SPDY and that RTT and bandwidth become more important.

obtain the dependency graphs of the top 200 Alexa Web pages [1]. `Epload` runs on a Mac with 2GHz dual core CPU and 4GB memory. We vary other factors based on the parameter space described in Table 1. Due to space limit, we only show figures under a 10Mbps bandwidth.

Figure 12 shows the performance of SPDY versus HTTP after incorporating dependencies and computation. Compared to Figure 6, dependencies and computation largely reduce the amount that SPDY helps or hurts. We make the following observations along with supporting evidence. First, computation and dependencies increase PLTs of both HTTP and SPDY, reducing the network load. Second, SPDY reduces the amount of time a connection is idle, lowering the possibility of slow start (see Figure 13). Third, dependencies help HTTP by making traffic less bursty, resulting in fewer retransmissions (see Figure 14). Fourth, having fewer outstanding objects diminishes SPDY's gains, because SPDY helps more when there are a large number of outstanding objects (as suggested by the decision tree in Figure 2). Here, we see that dependencies and computation reduce and can easily nullify the benefits of SPDY, implying that speeding up computation or breaking dependencies might be necessary to improve the PLT using SPDY.

Interestingly, we find that RTT and bandwidth now play a more important role in the performance of SPDY. For example, Figure 12 shows that SPDY helps up to 80% of the pages under low bandwidths, but only 55% of the pages under high bandwidths. This is because RTT and bandwidth determine the amount of time page loads spend in network relative to computation, and further the

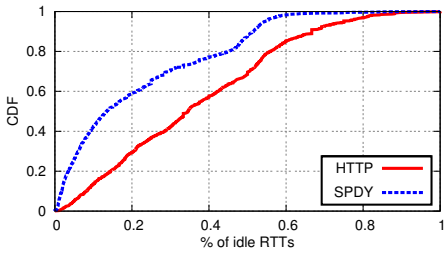



Figure 13: Fractions of RTTs when a TCP connection is idle. Experimented under 2% loss rate.

amount of impact that computation has on SPDY. This explains why SPDY provides minimal improvements under good network conditions (see Figure 12(c)).

To identify the impact of computation, we scale the time spent in each computation activity by factors of 0, 0.5, and 2. Figure 15 shows the performance of SPDY versus HTTP, both with scaled computation and under high bandwidths, suggesting that speeding up computation increases the impact of SPDY. Surprisingly, speeding up computation to the extreme is sometimes no better than a x2 speedup. This is because computation delays the requesting of dependent objects which allows for previously requested objects to be loaded faster, and therefore possibly lowers the PLT.

### 5.3 Advancing SPDY

SPDY provides two mechanisms, i) prioritization and ii) server push, to mitigate the negative effects of dependencies and computation of real page loads. However, little is known about how to better use the mechanisms. In this section, we explore advanced policies to speed up page

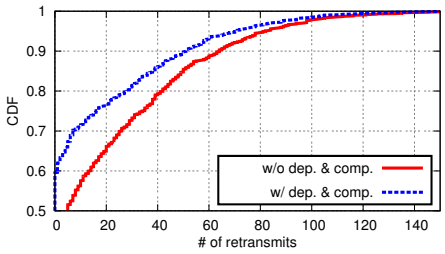



Figure 14: SPDY helps reduce retransmissions.

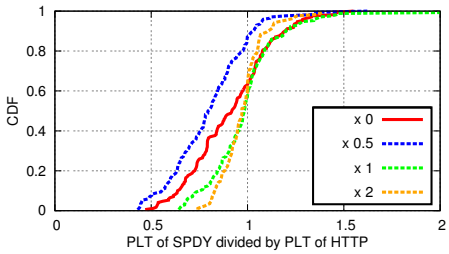



Figure 15: Results by varying computation when bw=10Mbps, rtt=200ms.

loads using these mechanisms.

### 5.3.1 Basis of advancing

To better schedule objects, both prioritization and server push provide mechanisms to specify the importance for each object. Thus, the key issue is to identify the importance of objects in an automatic manner. To highlight the benefits, we leverage the dependency information obtained from a previous load of the same page. This information gives us ground truth as to which objects are critical for reducing PLT. For example, in Figure 10, all the activities depend on loading the HTML, making HTML the most important object; but no activity depends on loading the image, suggesting that the image is not an important object.

To quantify the importance of an object, we first look at the time required to finish the page load starting from the load of this object. We denote this as time to finish (TTF). In Figure 10, TTF of the image is simply the time to load the image alone, while TTF of JS2 is the time to both load and evaluate it. Because TTF of the image is longer than TTF of JS2, this image is more important than JS2. Unfortunately in practice, it is not clear as to how long it would take to load an object, before we make the decision to prioritize or push it.

Therefore, we simplify the definition of importance. First, we convert the activity-based dependency graph to an object-based graph by eliminating computation while preserving dependencies (Figure 16). Second, we calculate the longest path from each object to the leaf objects; this process is equivalent to calculating node depths of a directed acyclic graph. Figure 16 (right) shows an example of assigned depths. Note that the depth here equals

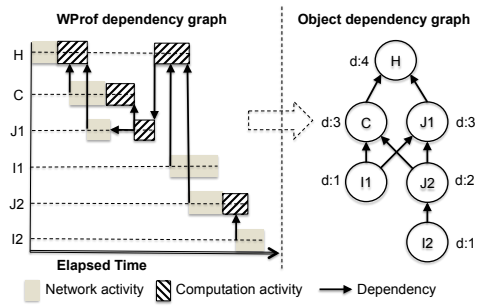
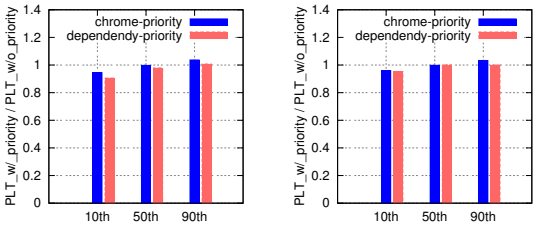



Figure 16: Converting WProf dependency graph to an object-based graph. Calculating a depth to each object in the object-based graph.




(a) rtt=20ms, bw=10Mbps (b) rtt=200ms, bw=10Mbps

Figure 17: Results of priority (zero packet loss) when bw=10Mbps. bw=1Mbps results are similar to (b).

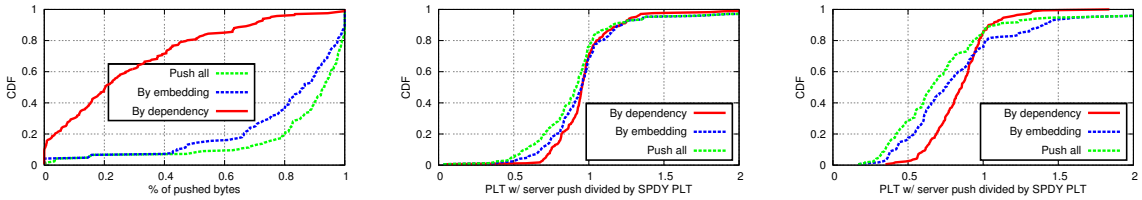
TTF if we ignore computation and suppose that the load of each object takes the same amount of time.

We use this depth information to prioritize and push objects. This implies that the browser or the server should know this beforehand. We provide a tool to let Web developers measure the depth information for objects transported by their pages.

### 5.3.2 Prioritization

SPDY/3 allows eight priority levels for clients to use when requesting objects. SPDY best practices website [18] recommends prioritizing HTML over CSS/JavaScript and CSS/JS over the rest (`chrome-priority`). Our priority levels are obtained by linearly mapping the depth information computed above (`dependency-priority`).

We compare the two prioritization policies to baseline SPDY in Figure 17. Interestingly, we find that there is almost no benefit by using `chrome-priority` while `dependency-policy` marginally helps under a 20ms RTT. The impact of *explicit* prioritization is minimal because the dependency graph has already *implicitly* prioritized objects. Implicit prioritization results from browser policies, independent of Web pages themselves. For example in Figure 10, all other objects cannot be loaded before HTML; Image and JS2 cannot be loaded before CSS and JS1. As dependencies limit the impact of SPDY, prioritization cannot break dependencies, and thus is unlikely to improve SPDY's PLT.




(a) Pushed bytes (b) rtt=20ms, bw=10Mbps (c) rtt=200ms, bw=10Mbps

Figure 18: Results of server push when bw=10Mbps.

### 5.3.3 Server push

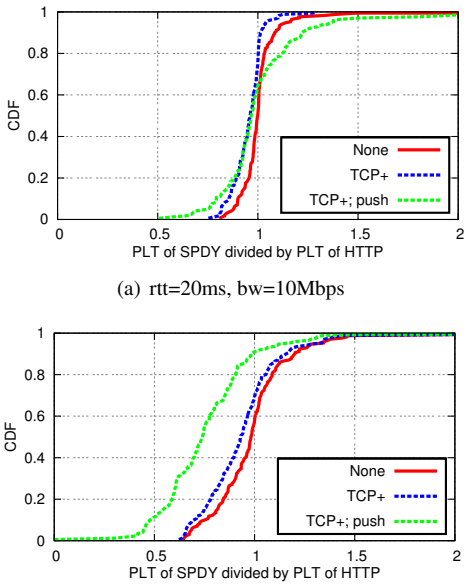
SPDY allows servers to push objects to save round trips. However, server push is non-trivial because there is a tension between making page loads faster and wasting bandwidth. Particularly, one should not overuse server push if pushed objects are already cached. Thus, the key goal is to speed up page loads while keeping the cost low.

We find no standard or best practices guidance from Google on how to do server push. Mod_spdy can be configured to push up to an *embedding level*, which is defined as follows: the root HTML page is at embedding level 0; objects at embedding level $i$ are those whose URLs are embedded in objects at embedding level $i-1$. An alternative policy is to push based on the depth information.

Figure 18 shows server push performance (i.e., push all objects, one embedding level, and one dependency level) compared to baseline SPDY. We find that server push helps, especially under high RTT. We also find that pushing by dependency incurs comparable speedups to pushing by embedding, while benefiting from a 80% reduction in pushed bytes (Figure 18(a)). Note that server push does not always help because pushed objects share bandwidth with more important objects. In contrast to prioritization, server push can help because it breaks dependencies which limits the performance gains of SPDY.

### 5.4 Putting it all together

We now pool together the various enhancements (i.e., TCP+ and server push by one dependency level). Figure 19 shows that this improves SPDY by 30% under high RTTs. But this improvement largely diminishes under low RTTs where computation dominates page load times.




(a) rtt=20ms, bw=10Mbps

(b) rtt=200ms, bw=10Mbps

Figure 19: Put all together when bw=10Mbps.

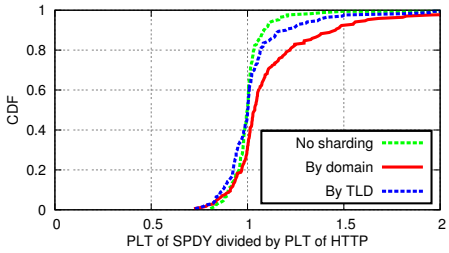



Figure 20: Results of domain shading when bw=10Mbps and rtt=20ms.

## 6 Discussions

**SPDY in the wild:** To evaluate SPDY in the wild, we place clients at Virginia (US-East), North California (US-West), and Ireland (Europe) using Amazon EC2 micro-instances. We add explanatory power by periodically probing network parameters between clients and the server, and find that RTTs are consistent: 22ms (US-East), 71ms (US-West), and 168ms (Europe). For all vantage points, bandwidths are high (10Mbps to 143Mbps) and loss rates are extremely low. These network parameters well explain our SPDY evaluations in the wild (not shown due to space limit) that are similar to synthetic ones under high bandwidths and low loss rates. The evaluations here are preliminary and covering a complete set of scenarios would be future work.

**Domain sharding:** As suggested by SPDY best practices [18], we used a single connection to fetch all the objects of a page to eliminate the negative impact of domain sharing. In practice, migrating objects to one domain suffers from deployment issues given popular uses of third parties (e.g., CDNs, Ads, and Analytics). To this

end, we evaluate situations when objects are distributed to multiple servers that cooperatively use SPDY. We distribute objects by full domain to represent the state-of-the-art of domain sharding. We also distribute objects by top-level domain (TLD). This demonstrates the situation when websites have eliminated domain sharding but still use third-party services. Figure 20 compares SPDY performance under these object distributions. We find that domain sharding hurts as expected but hosting objects by TLD is comparable to using one connection, suggesting that SPDY's performance does not degrade much when some portions of the page are provided by third-party services.

**SSL:** SSL adds overhead to page loads which can degrade the impact of SPDY, but it keeps the handshake overhead low by using a single connection. We conduct our experiments using SSL and find that the overhead of SSL is too small to affect SPDY's performance.

**Mobile:** We perform a small set of SPDY measurements under mobile environments. We assume large RTTs, low bandwidths, high losses, and large computational delays, as suggested by related literature [3, 26]. Results with simulated slow networks suggest that SPDY helps more but also hurts more. It also shows that prioritization and server push by dependency help less (not shown due to space limit). However, large computational delays on mobile devices reduce the benefits provided by SPDY. This means that the benefits of SPDY under mobile scenarios depends on the relative changes in performance of the network and computation. Further studies on real mobile devices and networks would advance the understanding in this space.

**Limitations:** Our work does not consider a number of aspects. First, we did not evaluate the effects of header compression because it is not expected to provide significant benefits. Second, we did not evaluate dynamic pages which take more time in server processing. Similar to browser computation, server processing will likely reduce the impact of SPDY. Last, we are unable to evaluate SPDY under production servers where network is heavily used.

## 7 Related Work

**SPDY studies:** Erman et al. [7] studied SPDY in the wild on 20 Web pages by using cellular connections and SPDY proxies. They found that SPDY performed poorly while interacting with radios due to a large body of unnecessary retransmissions. We used more reliable connections, enabled SPDY on servers, and swept a more complete parameter space. Other SPDY studies include the SPDY white paper [20] and measurements by Microsoft [14], Akamai [13], and Cable Labs [19]. The SPDY white paper shows a 27% to 60% speedup for SPDY, but the other studies show that SPDY helps only marginally. While providing invaluable measurements, these studies look at a limited parameter space. Studies by Microsoft [14] and Cable Labs [19] only measured single Web pages and the other studies consider only a limited set of network conditions. Our study extensively swept the parameter space including network parameters, TCP settings, and Web page characteristics. We are the first to isolate the effect of dependencies, which are found to limit the impact of SPDY.

**TCP enhancements for the Web:** Google have proposed and deployed several TCP enhancements to make the Web faster. TCP fast open eliminates the TCP connection setup time by sending application data in the SYN packet [15]. Proportional rate reduction smoothly backs off congestion window to transmit more data under packet loss [5]. Tail loss probe [23] and other measurement-driven enhancements described in [8] mitigated or eliminated loss recovery by retransmission timeout. Our TCP modifications are specific to SPDY and are orthogonal to Google's proposals.

**Advanced SPDY mechanisms:** There are no recommended policies on how to use the server push mechanism. We find that mod_spdy [11] implements server push by embedding levels. However, we find that this push policy wastes bandwidths. We provide a server push policy based on dependency levels that performs comparably to mod_spdy's while pushing 80% less data.

## 8 Conclusion

Our experiments and prior work show that SPDY can either help or sometimes hurt the load times of real Web pages by browsers compared to using HTTP. To learn which factors lead to performance improvements, we start with simple, synthetic page loads and progressively add key features of the real page load process. We find that most of the performance impact of SPDY comes from its use of a single TCP connection: when there is little network loss a single connection tends to perform well, but when there is high loss a set of connections tend to perform better. However, the benefits from a single TCP connection can be easily overwhelmed by dependencies in real Web pages and browser computation. We conclude that further benefits in PLT will require changes to restructure the page load process, such as the server push feature of SPDY, as well as careful configuration at the TCP level to ensure good network performance.

## Acknowledgements

We thank Will Chan, Yu-Chung Cheng, and Roberto Peon from Google, our shepherd, Sanjay Rao, and the anonymous reviewers for their feedback. We thank Ruhui Yan for helping analyze packet traces.

## References


[1] Alexa - The Web Information Company. http://www.alexa.com/topsites/countries/US.

[2] Data compression in chrome beta for android. http://blog.chromium.org/2013/03/data-compression-in-chrome-beta-for.html.

[3] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[4] National Broadband Map. http://www.broadbandmap.gov/.

[5] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional Rate Reduction for TCP. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[6] Dummynet. http://info.iet.unipi.it/~luigi/dummynet/.

[7] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDYier Mobile Web? In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2013*.

[8] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *Proc. of the ACM Sigcomm, 2013*.

[9] T. J. Hacker, B. D. Noble, and B. D. Athey. The Effects of Systemic Packet Loss on Aggregate TCP Flows . In *Proc. of IEEE Iternational Parallel and Distributed Processing Symposium (IPDPS), 2002*.

[10] HTTP/2.0 Draft Specifications. https://github.com/http2/http2-spec.

[11] mod_spdy. https://code.google.com/p/mod-spdy/.

[12] World Flags mod_spdy Demo. https://www.modspdy.com/world-flags/.

[13] Not as SPDY as you thought. http://www.guypo.com/technical/not-as-spdy-as-you-thought/.

[14] J. Padhye and H. F. Nielsen. A comparison of SPDY and HTTP performance. In *MSR-TR-2012-102*.

[15] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

[16] Amazon silk browser. http://amazonsilk.wordpress.com/.

[17] Chapter 11. HTTP 1.X. http://chimera.labs.oreilly.com/books/1230000000545/ch11.html.

[18] SPDY best practices. http://dev.chromium.org/spdy/spdy-best-practices.

[19] Analysis of SPDY and TCP Initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.

[20] SPDY whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[21] SPDY protocol–Draft 3. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3.

[22] Spdylay - SPDY C Library. https://github.com/tatsuhiro-t/spdylay.

[23] Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[24] W3C DOM Level 3 Events Specification. http://www.w3.org/TR/DOM-Level-3-Events/.

[25] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2013*.

[26] X. S. Wang, H. Shen, and D. Wetherall. Accelerating the Mobile Web with Selective Offloading. In *Proc. of the ACM Sigcomm Workshop on Mobile Cloud Computing (MCC), 2013*.