Latency Amplification: Characterizing the Impact of Web Page Content on Load Times

Technical Report CS-TR-2013-11
Cheriton School of Computer Science
University of Waterloo

February 1, 2013

Cătălin Avram, *University of Waterloo*
Kenneth Salem, *University of Waterloo*
Bernard Wong, *University of Waterloo*

Abstract

Web users like sites that load quickly. Longer web page load times translate to reduced user satisfaction and loss of revenue and mindshare. The time required to load a given web page is difficult to predict because it is a complex function of many factors, such as the latencies associated with the network requests used to retrieve that content from remote servers. However, one of the most important factors is the page content, including the scripts, images, style sheets and other objects that are present on the page. In this paper we propose a simple metric for characterizing the content of a web page in terms of its impact on page loading times. This metric, called the latency amplification factor (LAF), characterizes the content of a web page in terms of how it affects the page load time. The LAF of a web page can be estimated quickly and easily, and we describe a lightweight method for doing so. In addition, we propose an extended version of the basic LAF metric, called CLAF, that relates page load time to underlying request latencies in the presence of content delivery networks. We estimated LAFs for a variety of popular web sites, and found that they varied substantially. To validate our approach for estimating LAFs, we compared estimated LAFs against measured LAFs and found that our methodology, though simple, gave reasonably accurate estimates.

1 Introduction

The web has had a tremendous impact on many parts of our daily lives. It has enabled us to have instant access to vast amounts of information. By virtually connecting all of us together, it has changed not only how we communicate, work, and purchase goods and services, but also the pace at which we do these things. A perhaps unsurprising result of this increased pace is that, unlike traditional customers at bricks-and-mortar stores, many web users expect to have near instant access to online services. Several recent studies [6, 8] have found that even a small increase in web page load times can substantially increase the likelihood that a customer would switch to a competing service or store. Therefore, reducing web page load time is critically important to the success of any web-based company, and it is important to understand what contributes to this time.

There are many factors that affect load times. These factors include, for example, the HTML and Javascript parsing and processing time within the browser and the network latency between clients and servers. However, one of the most important factors affecting page load time is the richness and complexity of the page content. Loading and rendering a single modern web page may require that the client download tens or hundreds of objects from multiple servers: these objects may include scripts, style sheets, images, and many other types of content. Furthermore, the structure of modern web pages often introduces object loading dependencies, where certain objects are only retrieved after another object has been retrieved and, if the object is HTML or Javacript, parsed and/or executed. Therefore, object loading dependencies introduce additional network round-trip delays, which can dominate the page load time.

Given the importance of page load times to user satisfaction and the strong impact of page content on load times, it is important to be able to understand and characterize that content. In this paper, we take a step in this direction by proposing a simple content metric - a means of characterizing the content of a web page and the impact of that content on page load times. One possible approach to this task would be to use some kind of "syntactic" metric, such as a count of the number of component objects on the page. However, since our real interest lies in the effect of page content on the page load time, the metric that we propose - which we call the *latency amplification factor (LAF)*, is behavioral rather than syntactic.

The LAF for a web page can be interpreted as the answer to a simple hypothetical question: if the individual request latencies between the client and the servers that hold the web page content were to increase by a constant factor, by what factor would the total load time for the web page increase? In other words, the LAF of a page is a measure of the sensitivity of a page's load time to changes in underlying request latencies for the objects on that page. A simple, small, plain HTML page that did not have any component objects would have a LAF of 1. More complex pages, with many interdependent objects, will have higher LAFs. A key feature of this metric is that it is simple and easy to compute.

This paper makes several contributions. First, we introduce the LAF metric, which captures the relationship between network latency and web page load time as a single number. Second, we present a lightweight methodology for estimating the LAF for a given web page. Our methodology estimates the LAF using a page retrieval log that can be generated easily by off-the-shelf browsers, and we've used it to measure the LAFs for a variety of a pages from popular web sites. Third, we have validated the LAFs we estimated. To validate the LAFs, we compare them to measured increases in page load time as we systematically inject a controlled amount of synthetic latency between the downloading client and the content servers. Finally, we consider a extended version of the LAF metric, which we refer to as *core LAF*, or CLAF. This metric distinguishes objects retrieved from content delivery networks (CDNs) from those fetched from non-CDN, or core, servers. CLAF is an estimate of the sensitivity of page load time to changes in the retrieval times of core objects (only). By comparing a page's LAF and CLAF, we can characterize how effective the CDN is at reducing the page's load time.

The rest of the paper is structured as follows. Section 2 provides some background, and illustrates the complexity of loading a web page with rich content. Section 3 introduces the LAF metric, and explains how we estimate the LAF of a given web page. We have used this methodology to estimate LAFs for a variety of popular web sites, and we present these results in Section 4. Section 5 presents the results of an empirical validation that compares LAFs estimated using our methodology with measured LAFs. In Section 6 we show a simple method for generalizing the LAF metric to account for effects of content delivery networks (CDNs). Section 7 summarizes related work, and Section 8 concludes.

2 Web Latency

Modern web pages are rich and complex. Each HTML, Javascript, and CSS object on a website may reference tens or hundreds of other objects. Thus, a browser loading a web page will have to issue many server requests to retrieve those objects. Furthermore, the presence of inter-object references, or dependencies, means that the browser must retrieve and parse the parent object, at least partially, before it can determine the objects that the parent refers to and issue requests for those objects.

We will use a very simple example web page to illustrate this process. The web page displays a single user-clickable button over a background image. As the button is clicked by the user, the background rotates through a sequence of three possible images.

Loading a web page generally begins with the user initiating a page load by clicking a link or manually entering an URL. The browser then retrieves the requested web page. In this example, the user requests the page `http://www.example.com/example.html`, the contents of which are shown in Figure 1. We will assume that browser has previously resolved the `example.com` domain name, and so can immediately fetch the requested web page from the remote server.

```
<html lang="en">
<head>
  <link rel="stylesheet" href="example.css"/>
  <script src="http://code.jquery.com\
/jquery-git.js"></script>
  <script src="http://code.jquery.com\
/ui/jquery-ui-git.js"></script>
  <script src="example.js"></script>
</head>
<body>
  <button>Change Background</button>
</body>
</html>
```

Figure 1: The web page `example.html`

Once the user's browser downloads and parses the root object (`example.html`), it will be able to identify that it includes references to a CSS stylesheet and three different script files. (Some modern browsers, such as Firefox and Chrome, may start parsing the root object on the fly, before the object has been fully downloaded.) As the browser parses the page, it begins rendering different parts of it. In addition, if the browser encounters references to other objects, such as `example.css` and `example.js`, it can issue asynchronous requests to retrieve those objects. Note that the browser must download and parse at least part of `example.html` before it can recognize the dependencies on other objects.

The stylesheet (`example.css`) is shown in Figure 2. It is responsible for loading a background image for the web page. The stylesheet is dependent on an image file (`fruits-01.jpg`), which the browser must retrieve by issuing another asynchronous request. This example also illustrates that the various objects on which the root page (`example.html`) depends may come from a variety of different servers.

```
body {
  background-image:url('http://www.free\
-pictures-photos.com/fruits/fruits-01.jpg');
  background-repeat:no-repeat;
}
```

Figure 2: The stylesheet `example.css`

The root web page also references the three script files. The first two (`jQuery` and `jQueryUI`) are well known JavaScript libraries that are used by many website developers. The third script, `example.js`, which is shown in Figure 3, makes use of those libraries. It attaches an event listener to the button on the page, so that clicking the button will cause the background image to change.

```
var bgNumber = 0;
$(function() {
  $("button")
    .button()
    .click(function( event ) {
      bgNumber = (bgNumber + 1) % 3;
      $("body").css("background-image",
        "url(http://www.free-pictures-" +
        "photos.com/fruits/fruits-0" +
        (bgNumber + 1) + ".jpg)");
    });
});
```

Figure 3: The script `example.js`

In this paper, we are primarily interested in the *load time* of a web page. We define this to include the time to load the root web page as well as any additional objects on which the root object depends, either directly or indirectly. It is not uncommon for complex web pages to request additional objects after the page load time. This may occur, for example, in response to additional user actions, or timers in background scripts. In our example, such a request would occur when the user uses the button on the web page to cycle to the next background image. Nevertheless, we consider a web page's load time to be the time from the initial user request for the root page until the time when all of its dependent objects have been fully retrieved, as the page is fully ready to use by the user once this occurs.

As the richness and complexity of a web page increases, we expect that the load time for that page will also increase. Even our very simple example requires fetching CSS, script, and image files before the page can be considered to be loaded. As we will see, popular web pages may be significantly more complex than our example, with many more objects and longer dependency chains.

3 Latency Expansion

Our objective is to arrive at a metric to characterize the content of a web page in terms of its effect on the page's loading time. Previous work, such as WebProphet [7], has focused on building models that can predict the absolute load time for a web page, taking the page content (and many other factors) into account. In contrast to such relatively complex models, our characterization cannot, by itself, be used to predict absolute page load times. Rather, we characterize web page content by considering how much the page structure magnifies the request latencies of the individual components that make up the page. As was introduced in Section 1, we call this the latency amplification factor (LAF) for the page.

The following simple thought experiment gives some intuition for the LAF. Suppose that a browser loads a web page P, and the load time for P is $t_P(0)$ Now suppose that the browser loads the same page again, but this time the retrieval latency for every component of the page is increased by a constant amount α. Suppose that the measured page load time for this second experiment is $t_p(\alpha)$. The latency amplification factor (LAF) for P is:

$$\frac{t_P(\alpha) - t_P(0)}{\alpha} \tag{1}$$

The numerator in this expression $(t_P(\alpha) - t_p(0))$ indicates the amount by which the page load time increased, while the denominator (α) indicates the amount by which the component latencies increased. For a simple web page with a single component, the latency amplification factor will be 1. For more complex pages that contain multiple dependent components, we expect to see amplification factors greater than one. The larger the LAF for page P, the more a change in underlying latency will affect the page load time experienced by the user.

In the remainder of this section, we present our methodology for estimating the LAF of a web page. This process requires two steps; First, we produce a *dependency graph* for the page. The dependency graph represents the structure of the page contents, i.e., its components, and the relationships among those components. The second step of our methodology is to analyze the page's dependency graph to estimate the page's LAF.

3.1 Dependency Graph

As was illustrated by the example in Section 2, loading a web page involves loading multiple objects, potentially from multiple servers. If clients could load all of these object concurrently, page load time would depend only on the time required to load the slowest object. Although modern browsers can and do load multiple objects concurrently and asynchronously, resource limits and browser settings, such as the maximum number of threads per page load, restrict the amount of concurrency. More importantly, the structure of the web page content itself may fundamentally limit the amount of concurrency. For example, consider the simple web page (example.html) in Figure 1. In that example, the objects example.css, jquery-git.js, and example.js cannot be retrieved until the browser has retrieved and parsed (at least the first part of) example.html. Similarly, the browser cannot retrieve fruits-01.jpg until it has retrieved example.css. We refer to such dependencies, which arise from the structure of web page content, as *structural dependencies*.

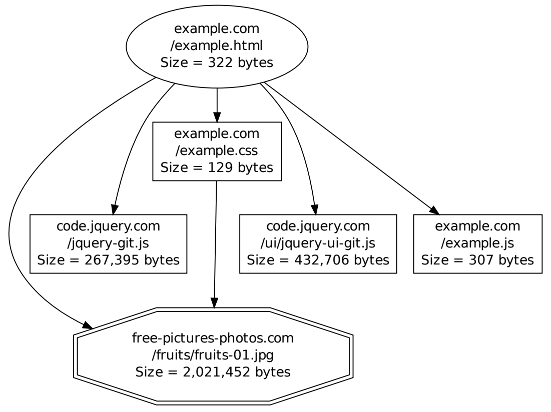

Figure 4: Dependency Graph for example.html from Section 2

We use a *dependency graph* to represent the structural dependencies of a given web page. A page's dependency graph contains one node for each retrievable object referred to, either directly or indirectly, by the given page. There is a directed edge $V_1 \rightarrow V_2$ in the dependency graph if the object V_1 refers to the object V_2. Dependency graphs are rooted directed graphs, with the root node representing the web page which the dependency graph represents. Figure 4 illustrates the dependency graph for the web page from Figure 1. Each node in the dependency graph is annotated with two attributes. The first is the size (in bytes) of the object represented by that node. The second is the site from which that object will be downloaded.

3.2 Producing Dependency Graphs

A number of different techniques can be used to generate a dependency graph for a given web page. One option, which we will refer to as the *white box* approach, is to analyze the web page content, along with the content of any dependent objects, to extract object references. This approach will correctly identify static references to objects. However, it requires the ability to parse and extract object references from all types of commonly used web objects. In our simple example, a white box approach must be able to parse HTML, JavaScript, and CSS objects. The main disadvantage of this approach is that, since it is based on static analysis, it will fail to detect dynamically-generated object references, which is very common especially within JavaScript objects.

Li et al [7] propose a dynamic, *black box* approach, embodied in their WebProphet system, to generate a dependency graph similar to the graph that we use. Their technique builds the dependency graph for a page by inspecting a log of object retrievals generated by the browser as it loads the page. Each log entry identifies a retrieved object, and provides some information about the timing of the retrieval, such as the retrieval request time and the request response time. To infer dependencies among the objects, WebProphet retrieves the page multiple times. On each retrieval, WebProphet uses a proxy-based technique to add artificial latency to the retrieval time of a target object from the trace. WebProphet then identifies other objects (in addition to the target) whose retrieval times are delayed, and infers that those objects are dependent on the target. In order to generate a close approximation of the entire dependency graph, WebProphet must repeat the latency injection process for a variety of latency injection targets, with multiple latency injections required for each target. This approach is also highly sensitive to latency fluctuations between measurements.

For our work, we instead use a lightweight black box approach to construct the dependency graph for a page. Like WebProphet, we use a browser to retrieve the web page for which we wish to construct the dependency graph, and then inspect the browser-generated request log. The log contains one entry for each object retrieved, with each entry containing the following information:

- the object's URL.
- the object's total size, in bytes.
- the *request time*, i.e., the time (relative to the beginning of the trace) at which the request for the object was sent by the browser.
- the *wait delay*, which is length of the interval between the request time and the receipt of the first byte of the object at the browser.

In particular, we use logs in the standard HTTP Archive (HAR) [11] format, which contains the necessary information and which can be generated natively by common browsers such as Chrome and Firefox.

We create dependency graphs with one node for each object referred to in the log, annotating each node with the object's size and site (extracted from the URL). We assume that object V_2 is dependent on object V_1, and create the corresponding directed edge from V_1 to V_2 in the graph, if and only if V_2's request time is after the end of the wait delay for V_1.

Clearly, this simple method of creating the dependency graph will identify all actual dependencies, both static and dynamic. However, it will also identify *false dependencies*, i.e., dependencies that do not reflect the structure of the web page. For example, an edge from V_1 to V_2 may exist in the graph simply because object V_2 was loaded after object V_1, and not because of an actual structural dependency between V_1 and V_2. On the other hand, this lightweight approach is considerably simpler than the black box approach used by WebProphet. That approach requires multiple page retrieval iterations and a means of injecting synthetic latency into object request times. In contrast, this simpler approach requires only a browser-generated retrieval log, which, in addition to being easier to collect, also facilitates performing multiple measurements to limit the impact of latency fluctuations.

3.3 Estimating the LAF

Latency amplification occurs because the browser must make multiple sequential round trips to retrieve objects. If the browser must make k network round trips to some site to retrieve content and the request latency for that site increases by an amount α, then the time the browser requires to retrieve all data from that site will increase by a factor $k\alpha$ - an amplification by k of change in request latency. Thus, to estimate a LAF from a dependency graph, we want to estimate how many sequential round trips are implied by that graph.

A very basic way to estimate the number of round trips implied by a dependency graph is to assume (conservatively) that any objects that can be retrieved concurrently will be retrieved concurrently. Furthermore, as a starting point we can assume that retrieving each object in the graph involves one network round trip. Under these simplifying assumptions, if there is a path of length k from root to leaf in the graph then the client will require at least k sequential round trips. Since the client is assumed to retrieve the objects along different paths in parallel, the LAF (the number of sequential round trips) would simply be the length of longest root-to-leaf path in the dependency graph.

Although this basic approach would give a simple approximation of the LAF, it fails to take into account a variety of relevant factors, such as the sizes of the objects, the properties of the TCP connections over which the object requests are issued, and the behavior of modern browsers. To estimate the LAF from the dependency graph, we therefore enhance the basic procedure to take these factors into account. Specifically, our LAF estimation procedure takes the following into account:

Object Size: The number of network requests required to retrieve an object depends on the size of the object – larger objects may require multiple round trips.

TCP Characteristics: The number of requests required to retrieve an object depends on the TCP window size, which is governed by TCP's congestion protocol. In addition, there is a fixed overhead associated with the TCP handshake required to establish a connection between the client and a site.

Connection Reuse: If the browser needs to retrieve multiple objects from the same site, it can use a single TCP connection to retrieve those objects, rather than establishing a new TCP connection for each requested object.

Our algorithm for estimating the LAF from a given dependency graph can be outlined as follows:

1. Estimate the number of round trips required to retrieve each individual node in the dependency graph.

2. For each distinct root-to-leaf path in the graph, count the total number of round-trips required along that path by summing the individual round trip counts for the nodes along the path.

3. Choose the maximum round-trip count over all root-to-leaf paths in the graph, and report that as the LAF.

To estimate the number of round trips required for each node, we use a simple model of the client's browser. Specifically, we assume that the browser will initiate the request to retrieve a node as soon as that node's predecessors have been retrieved, and that there is no limit on the number of concurrent requests that the browser may have outstanding. These assumptions will of course lead to inaccuracies in our model. However, they also ensure that the model is browser and machine independent, which is critical as most large-scale websites have very diverse clients. The number of round trips required to retrieve the object depends on the size of the object and on the state of the TCP connection (specifically, on the TCP window size) at the time the object is retrieved.

Consider a node in the graph with size s bytes, and suppose that the TCP window size for the connection used to retrieve the node is of size w_{start} bytes when retrieval starts. The value of s for each node can be found in the dependency graph, and we will explain shortly how

w_{start} is determined. We estimate r, the number of round trips required to retrieve this node, as:

$$r = \left\lceil \log_2 \left(\frac{s}{w_{start}} + 1 \right) \right\rceil \quad (2)$$

The factor of $\log_2$ in this expression arises from the fact that the TCP window size doubles with each round trip until the object has been completely retrieved. Thus, if $s \leq w_{start}$, the object will be retrieved in a single round trip ($r = 1$). If $s > w_{start}$, then $r > 1$ as more than one round trip will be needed. We can also define w_{finish}, which indicates what the TCP connection's window size will be after the node has been retrieved. If $s \leq w_{start}$, then no window size doubling will occur and w_{finish} will be the same as w_{start}. Otherwise, the window size will double one or more times during the retrieval of the node. Thus, the final window size can be determined using the following expression.

$$w_{finish} = 2^{r-1} w_{start} \quad (3)$$

where r is the round trip count for the node, as defined by Equation 2. This model assumes that objects are relatively small, and that most users will finish retrieving their objects before reaching their actual maximum window size.

The value of w_{start} for a node depends on which TCP connection is used to retrieve the object. The simple browser model that we use to estimate the LAF assumes that the browser will retrieve objects in parallel whenever possible, and that it will reuse existing TCP connections, which it can do whenever there are multiple sequential retrievals from the same site. Thus, a node's w_{start} will depend on whether that node has any ancestors in the dependency graph that are retrieved from the same site. We consider two cases. If there are no such ancestors, then we set $w_{start} = 4440$ bytes, which corresponds to TCP's default initial window size for new connections. On the other hand, if there are one or more such ancestors, then the browser will have one or more TCP connections already open to the target site. In this case, we conservatively set the node's w_{start} to the maximum w_{finish} among all of its same-site ancestors. This effectively assumes that the browser will choose the connection over which it has already retrieved the most data so far. This is a conservative choice because choosing the largest w_{finish} results in the smallest possible estimate for the number of round trips (Equation 2) for the current node, which in turn gives the smallest possible estimated LAF.

4 LAF Examples

We have focused on a set of ten popular web pages for testing, as summarized in Figure 5. Most of our choices appear among the top ten most visited web sites in the Alexa top 500 global sites ranking [1]. Our set includes a variety of different types of sites, including search engines, web portals, a social network, and e-commerce, news, and media delivery sites.

For each web page shown in Figure 5, we used the PhantomJS [12] web browser to retrieve the page while capturing a request log. This headless browser uses the WebKit layout engine, which also powers popular browsers such as Safari and Chrome. In all cases, the web client was located at the University of Waterloo, in Canada. Using these logs, we generate dependency graphs and then estimate the LAF for each page, as described in Section 3. We repeated this process 20 times for each web page, resulting in 20 estimated LAFs per page. For each web page in our test set, Figure 6 shows a boxplot representing the minimum, lower quartile, median, upper quartile and maximum values among the 20 LAF estimates for that page, along with the mean LAF computed over the 20 runs.

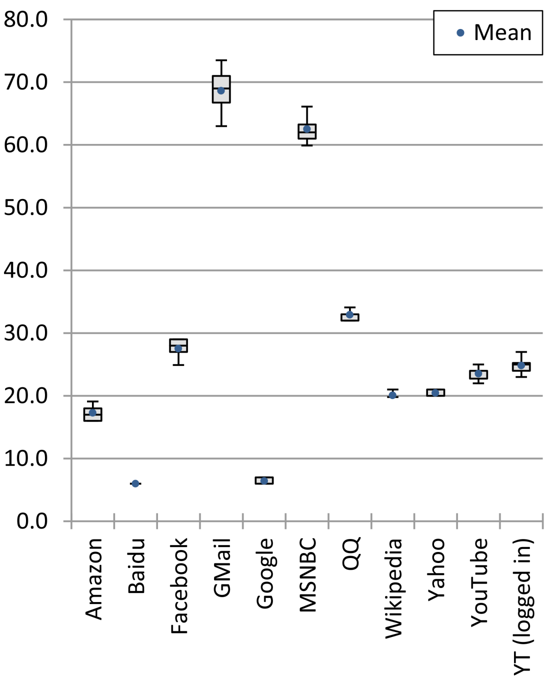

Figure 6: Boxplot of LAF estimates for various web pages.

In this experiment, we observed a wide range of LAFs from a low of 6 to a high of 75. Not surprisingly, the search engines (Google and Baidu), with their simple pages, had the lowest LAFs. At the other end of the spectrum were GMail (webmail) and MSNBC (news page), which had LAFs an order of magnitude higher than those of the search engines. Previous work [2] has suggested

Page	Description
Amazon `http://www.amazon.com/`	We used the public front page (no user has been logged in) of the popular shopping site. The web page includes a search box, product suggestions with images, and advertisements.
Facebook `https://www.facebook.com/`	We used the main timeline page of a logged-in user. The page contains several updates, comments and notifications from the user's friends, lists of applications, groups, pages and favorites, advertisements and many images. It also contains a chat application.
GMail `https://mail.google.com/mail`	We have measured the page showing the inbox of a logged-in user. The inbox contained 3500 e-mails out of which the first 100 are displayed. The account has been customized to include a theme with a large background image that changes based on the time of day. Other items on the page include a chat application, a calendar application displaying upcoming meetings and the weather, and several custom folders, two of which contain e-mail from different e-mail accounts being synchronized via the IMAP protocol.
Google `http://www.google.com/`	The main page of the popular search engine. This page is relatively sparse, with a logo, a textbox, and a few buttons and links. Google will redirect the browser to the localized version of the website, which in our case is `https://www.google.ca/`
Baidu `http://www.baidu.com/`	The main page of the popular Chinese search engine, which is served from Beijing. Like Google's main page, Baidu's main page is relatively sparse.
MSNBC `http://www.nbcnews.com`	A recent study [2] found that news websites load a significantly higher number of objects than others. We chose this page as a representative of such sites. It includes links and news highlights, followed by a large number of news sections, as well as many images and advertisements. This is the only website in our test set that is not among the top 10 in Alexa's global top sites ranking [1].
Wikipedia `http://en.wikipedia.org/wiki/Computer_science`	We used a specific article – the English version of the article on computer science – rather than the main landing page. The article includes a large number of images, some of which are GIF animations.
Yahoo `http://www.yahoo.com/`	We used the main Yahoo portal, which includes news, links, images, interactive widgets and advertisements.
QQ `http://www.qq.com/`	A popular Chinese portal site that is similar to Yahoo.
YouTube `http://www.youtube.com/`	We used the main landing page for this video site, both for an anonymous user and with a logged-in user account. In both cases, the page presents a list of video clips to watch, links, and a banner advertisement. For the anonymous user, the video recommendations are general, while a logged-in user sees customized recommendations, his/her avatar, and additional account-related information.

Figure 5: Tested web pages.

that news web pages often load objects from a large number of hosts. This was certainly true for the MSNBC front page, which has references to objects from almost 60 distinct hosts. The other websites we tested typically only reference about 10 hosts, and the search engines reference only 3. The large number of hosts translates to a lot of separate TCP connections, which, together with small initial congestion window size values, account for the large LAF value for the MSNBC page. In contrast to MSNBC, GMail's high LAF value is due to its high structural complexity, which is manifested as a very long critical path in its dependency graph.

The remaining sites we tested are between these extremes, with LAFs from just under 20 to just over 30. These LAFs, though much lower than those of GMail and MSNBC, are still very significant. For example, with a LAF near 30, a 30 ms change in network latency results in an almost 1 second change in user-perceived page load time, which is a substantial amount.

Figure 6 also shows that, with the possible exception of the GMail test, we did not see a wide variation among the 20 different LAFs that we estimated for each web page. Each LAF was generated using a different retrieval log, and we expect that different retrieval logs (for the same page) should include different request timing information because of natural variations in the request times and wait delays across the different runs. Nonetheless, such variations had only a small effect on the LAF estimates.

5 LAF Model Validation

In order to validate our LAF estimation methodology against empirical data, we measured web page loading times while manipulating the latency between our browser and the webservers. By comparing such measurements against baseline measurements with no added latency, we can determine the actual LAF of a web page. We then compare these measurements to the estimates made using the methodology from Section 3.

Suppose that $t_P(0)$ is the measured baseline page load time for a web page P, with no added latency. Similarly, suppose that $t_P(\alpha)$ is the page load time for P if we introduce an additional latency $\alpha/2$ to each incoming and outgoing network packet between the browser and the webservers, for a total additional round trip latency of α. The *measured* LAF for P can then be calculated using Equation 1.

To introduce network latency, we used the network traffic control tool (`tc`), which has been included in the Linux kernel since version 2.2. Since `tc` is only capable of manipulating outgoing traffic, we adopted a technique described by Nussbaum and Richard [10] to allow us to add latency to both incoming and outgoing traffic. Their technique makes use of a dummy network device, the `ifb` (intermediate functional block) device, to manipulate incoming traffic. All incoming traffic is redirected through the `ifb` device, and `tc` is used to add latency to outgoing traffic on both the physical network device and the `ifb` device. All of our measurements, including the baselines, were taken using this setup. For the baseline measurements, incoming traffic was routed through the `ifb` device, but no latency was added in either direction.

Figure 7 shows a comparison of measured and estimated LAFs, with $\alpha = 100$ms. for the measured LAFs. We repeated each page load measurement 20 times. For each web page we measured, the figure shows the mean estimated LAF, the mean measured LAF, and a 90% confidence interval around the measured LAF. The mean estimated LAFs are the same as those that were presented in Figure 6.

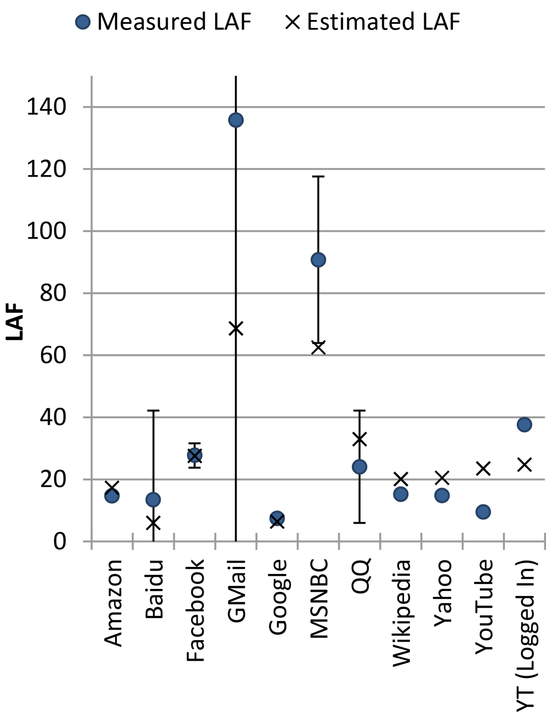

Figure 7: Mean measured and estimated LAF for different web pages. Error bars show the 90% confidence interval around the measured mean.

For most of our test pages the estimated LAF was reasonably close to the measured value, which suggests that our estimation methodology is capturing the important factors that contribute to page load times. For several sites, including Baidu, QQ, MSNBC and GMail, there was significant variance in measured page load times, resulting in relatively large confidence intervals around the mean measured LAF. For Baidu and QQ, both of which include a substantial amount of content served from China, this is due to the relatively long network path from those servers to our measurement point in Canada. MSNBC's content is served from closer sites, but as we have previously noted, that page includes content from a large number of servers, delays to any one of which can affect our measured page load times. GMail, which had an extremely large confidence interval, was the most problematic site from our test set. GMail uses a single long request to download a substantial amount of data in the background to the client. Although the GMail web page is usable well before these data are fully downloaded, our measured page load time includes the time for this request, which varies substantially from run to run. In addition to the substantial variance it introduces into our measured LAF, such requests also highlight a challenge for our LAF estimation methodology. In estimating the LAF, we include all objects on which the root page depends. However, the page may be usable well before all of those objects have been retrieved.

To measure the LAFs shown in Figure 7, we added 100 ms. of latency to each network round trip. Figure 8 generalizes this comparison by measuring LAFs for test web pages as we vary the amount of added latency from 50ms to 400ms. For each test site, Figure 8 shows the mean measured latency (with a 90% confidence interval), as well as our LAF estimate. Because of the high variance in latency measurements for Baidu, GMail, and QQ (as shown in Figure 7), have not included those sites in Figure 8, focusing instead on the test sites for which we can obtain accurate LAF measurements.

Figure 8 shows that, unlike our LAF estimates, measured LAFs are not independent of the amount of added latency. This is not surprising, as our procedure for estimating the LAF is based on a highly simplified model of browser and network behavior. The LAF is a simple measure of the impact of rich web page content on page load times. Our results suggest that for most sites (Facebook is an exception), measured LAFs decline as the amount of added latency increases, and appear to gradually level off. This suggests that the page load time penalty is not perfectly proportional to the amount of added request latency, as hypothesized by our simple LAF model. Rather, the page load time penalty is proportional to added request latency only when the request latencies are relatively large.

6 Accounting for CDNs

We estimate a page's LAF by asking what would happen to the page load time if all of the objects on which a page depends took longer to retrieve. However, in some cases, we may wish to focus only on certain objects, and ask how sensitive the page's load time is to the retrieval time of those objects. For example, many web service providers make use of content delivery networks (CDNs) to cache static page content at the edge of the network, closer to end users. For a web page from such a provider, we may wish to characterize only that part of the page that is not served by the CDN.

It is very easy to modify our methodology to answer such questions. In the remainder of this section, we show how to estimate the *core latency amplification factor (CLAF)* of a page. The CLAF is similar to the LAF, but it only considers the effect of page content that is not served by CDNs. A page's LAF can be interpreted as an estimate of the number of sequential network round trips that will be required to load that page, while its CLAF can be interpreted as an estimate of the number of sequential network round trips to core (non-CDN) servers. Comparing the LAF and CLAF of a page provides a measure of the CDN's effectiveness for that page.

To estimate the CLAF, we need to be able to identify which objects are served from CDNs and which are not. After generating the dependency graph for a web page as described in Section 3.2, we add an additional graph post-processing step that tags each node according to whether or not it is served from a CDN. We use a simple pattern matching test to determine whether an object is served from a CDN: we query the DNS name servers (using `dig`) for information about the host part of an object's URL and look for certain key words in the answer section of the DNS reply. For our example in Figure 1, we would use `dig example.com`. If the answer section contains any reference to well known CDNs such as "akamai" or "cloudfront" or if it simply contains the character sequence "cdn", we assume the host is indeed a CDN. Otherwise, we consider it a core host. The method is by no means comprehensive, but it provides a simple approach to approximate the impact of CDNs on the amplification factor.

Once the nodes have been tagged, we use a slight modification of the LAF estimation algorithm described in Section 3.3 to analyze the graph and estimate the page's CLAF. CLAF estimation is identical to LAF estimation except that the number of network round trips for CDN nodes is taken to be zero. Round trip counts for non-CDN nodes are estimated using Equation 2 as is the case when estimating the LAF.

Figure 9 shows both the CLAF and LAF for several sites. The five sites presented are the only ones that we have identified as using CDNs using our basic approach. For example, although we know that the websites under Google's administration (Google, GMail and YouTube) make use of CDNs, our pattern matching approach is unable to identify the servers responsible for these services.

As expected, Figure 9 shows that CLAF values are consistently lower then the LAF. For websites like Amazon and Facebook, the reduction is significant; for both

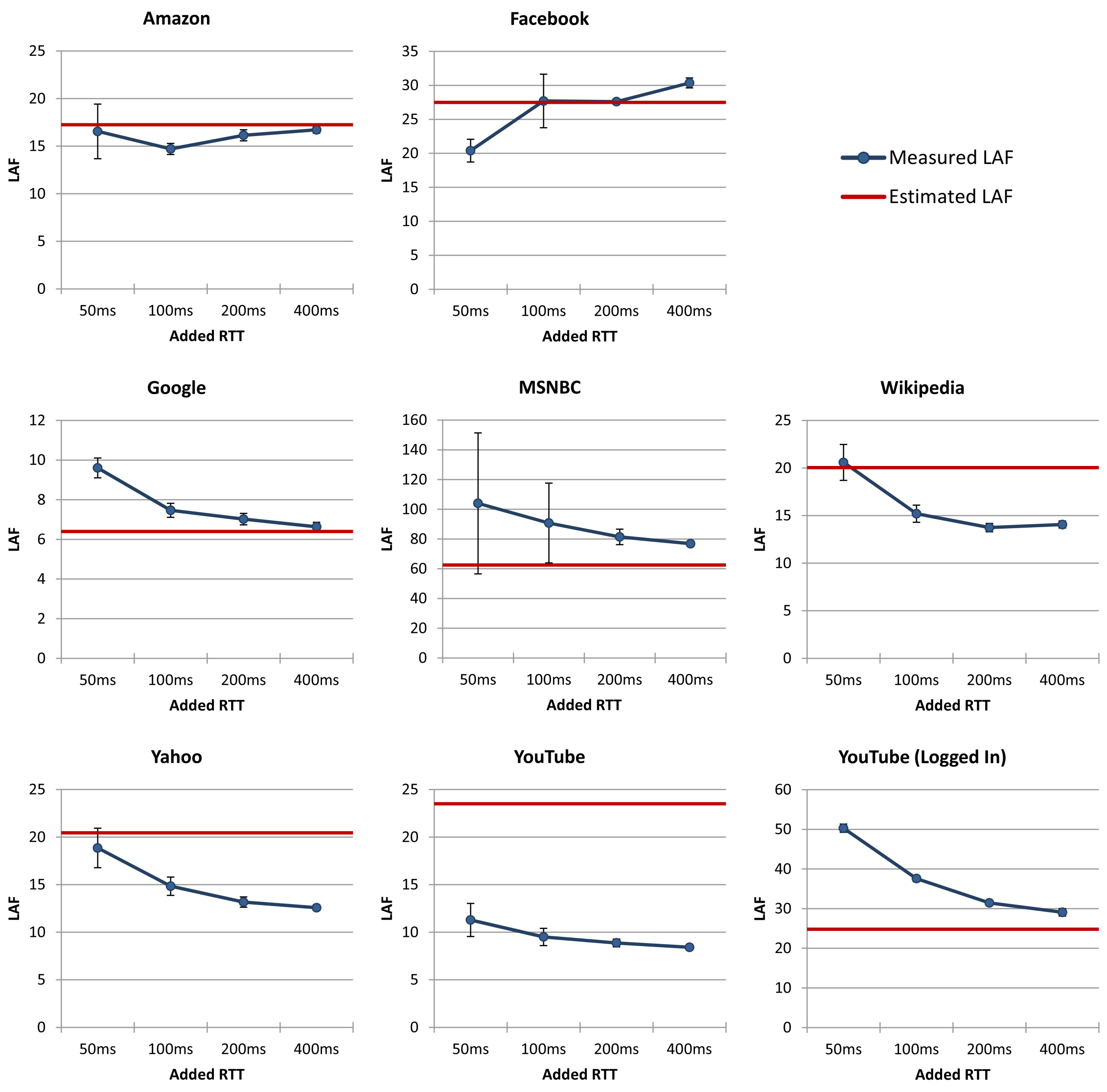

Figure 8: Measured LAF as a function of added latency

websites, all but one host are CDN nodes. With MSNBC and QQ, a one third to one half of the hosts have been identified as CDNs, so the reduction is more moderate. For these two sites however, even the lower CLAF values are still quite high.

In the case of Yahoo, our methodology identified only 2 CDN hosts out of the 12 hosts it uses. However neither of the associated nodes were on a critical path when calculating the LAF, so their omission from the CLAF calculation did not affect the final value. It is however possible that, in this case, there are other CDN hosts we have not been able to identify with our approach.

7 Related Work

Butkiewicz et al [2] have presented a detailed analysis of the complexity of modern web pages. This analysis, which included more than 1700 web sites, considers how web page features, such as the number of objects fetched, the types of content fetched, and the number of sources from which content is fetched, are related to the time required to render the page. This analysis also introduces a regression-based model that can be used to predict the rendering time of a page given values for key page features of the page content. The model we present in this paper is complementary to this work. While our models are sensitive to the content of the page, they predict the

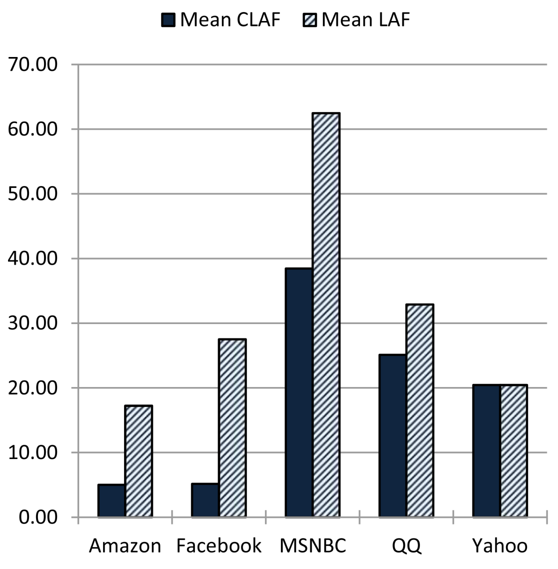

Figure 9: LAF and CLAF for Different Web Pages

impact of individual request latencies on the page loading time.

Because of the importance of web performance, a variety of tools and techniques are available for web performance analysis. For on the server side, there are tools available to help with performance analysis and server stress testing. These include commercial solutions, such as those offered by Neustar [9] or HP's Performance Center [5], as well as open-source projects like Benchlab [3]. These tools allow for the definition and execution application-specific, controllable, synthetic workloads so that servers can be tested under a variety of realistic load conditions.

For measuring client-side page load times, Rajamony and Elnozahy [13] present a technique that embeds measurement JavaScript directly in a downloaded web page. After measuring client side latencies, the embedded JavaScript then uploads measurements to a centralized repository. This approach can be employed by a web service provider to measure client-perceived page latencies for its web pages without any need for direct client involvement. A disadvantage of the approach is that it changes the behavior of the web page because of the introduction of the embeded Javascript. Wei and Xu [15] describe an alternative technique for estimating client-perceived page load latencies. Their technique relies on the ability to monitor network traffic into and out of a web service. Because it only inspects SSL headers, this approach is non-intrusive: no changes are required in the web service or on the client side, and there is no need to inject measurement code into the web page content. In contrast to these approaches, our trace-based approach is browser-specific and does require the client's involvement for trace collection. However, it does not require any changes to web page contents. Unlike the technique proposed by Wei and Xu, our client-centric approach can easily account for all of the latencies that clients experience when loading pages that include objects from multiple sources. Furthermore, we can directly measure client latencies, rather than inferring them from network packet observations.

Both of the previous techniques seek to measure client-perceived page download times. In contrast, the objective of the WebProphet system [7] is to predict the load time of a web page under hypothetical conditions. WebProphet models a page using a graph, with edges representing dependencies among the objects on the page. To infer dependencies, WebProphet retrieves the page while adding latency to the download of specific objects. By identifying which other objects' latencies change as a result of these injections, WebProphet can infer that dependencies exist. Ultimately, WebProphet produces a very rich model of web page load time, with many input parameters. To make "what if" predictions about page load times, WebProphet uses its model to simulate the loading of the web page under hypothetical conditions. With this approach, WebProphet can predict the effects of browser configuration changes, changes in the retrieval latency of individual objects or groups of objects, changes in network-level characteristics such as the time required for DNS resolution. In contrast, the technique proposed here produces a much simpler model that is intended only to relate page load time to latencies of the requests for the objects on the page. Our model cannot, for example, be used to predict the effect of a change in browser configuration on page load time. Although WebProphet's model is considerably richer, this richness comes with a price: its models are larger, more expensive to build, and more expensive to use than the models proposed here. We are trying to answer a narrower but important question using a model that is simple and easy to use.

In addition to WebProphet, which, like our work, focuses on client-centric what-if latency analysis, there are also techniques for performing what-if performance analyses on the server side. The "What-If Scenario Evaluator" (WISE) [14] takes a machine learning approach to model the effects of changes in a web service's deployment configuration. For example, WISE was used to predict the effect on users' page load times when a CDN node serving those users was moved. WISE requires an extensive collection of monitoring data which it can use to infer dependencies among system parameters of interest, such as client response times and CDN node locations. Chen et al [4] focus on the effect of changes in the inter-tier latencies in a multi-tier service architecture.

They propose a metric called the *link gradient* to capture the effect of the latency between two tiers on the overall performance of the system.

8 Conclusion

Although user satisfaction depends strongly on the load time of a web page, there is no simple way to characterize the relationship between the content of a page and its load time. We have proposed a simple metric, the LAF, to address this problem. This metric is easy to estimate and to interpret. We have estimated LAFs for some of the world's most popular websites. Some are surprisingly high, indicating the small changes in network latencies can lead to large increases in page load times.

We validated our procedure for estimating LAFs by measuring LAFs and comparing those measurements to our estimates. In most cases, our LAF estimations were reasonably accurate. Finally, we have considered an enhanced metric called CLAF that can distinguish web page content loaded from CDNs from other content. Comparing the LAF and CLAF for a web page gives a way of quantifying CDN effectiveness for that page.

References

[1] ALEXA. Global top sites list. `http://www.alexa.com/topsites/global`.

[2] BUTKIEWICZ, M., MADHYASTHA, H. V., AND SEKAR, V. Understanding website complexity: measurements, metrics, and implications. In *Internet Measurement Conference* (2011), P. Thiran and W. Willinger, Eds., ACM, pp. 313–328.

[3] CECCHET, E., UDAYABHANU, V., WOOD, T., AND SHENOY, P. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development* (Berkeley, CA, USA, 2011), WebApps'11, USENIX Association, pp. 4–4.

[4] CHEN, S., JOSHI, K. R., HILTUNEN, M. A., SANDERS, W. H., AND SCHLICHTING, R. D. Link gradients: Predicting the impact of network latency on multitier applications. In *INFOCOM* (2009), IEEE, pp. 2258–2266.

[5] HP. Hp performance center. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1172026`.

[6] KOHAVI, R., AND LONGBOTHAM, R. Online experiments: Lessons learned. *Computer 40*, 9 (sept. 2007), 103 –105.

[7] LI, Z., ZHANG, M., ZHU, Z., CHEN, Y., GREENBERG, A. G., AND WANG, Y.-M. Webprophet: Automating performance prediction for web services. In *NSDI* (2010), USENIX Association, pp. 143–158.

[8] MAYER, M. What google knows. In *Proceedings of the Third Annual Web 2.0 Summit* (San Francisco, CA, USA, November 2006).

[9] NEUSTAR. Benefits of external load testing: Identify bottlenecks and improve customer experience. Tech. rep. available online at `http://www.neustar.biz/enterprise/resources/web-performance/benefits-of-external-load-testing#.UQlpnWQqa4V`.

[10] NUSSBAUM, L., AND RICHARD, O. A comparative study of network link emulators. In *SpringSim* (2009), G. A. Wainer, C. A. Shaffer, R. M. McGraw, and M. J. Chinni, Eds., SCS/ACM.

[11] ODVARKO, J. Http archive specification version 1.2. `http://www.softwareishard.com/blog/har-12-spec/`.

[12] PHANTOMJS. http://phantomjs.org/.

[13] RAJAMONY, R., AND ELNOZAHY, E. M. Measuring client-perceived response time on the www. In *USITS* (2001), USENIX.

[14] TARIQ, M. M. B., ZEITOUN, A., VALANCIUS, V., FEAMSTER, N., AND AMMAR, M. H. Answering what-if deployment and configuration questions with WISE. In *SIGCOMM* (2008), V. Bahl, D. Wetherall, S. Savage, and I. Stoica, Eds., ACM, pp. 99–110.

[15] WEI, J., AND XU, C.-Z. Measuring client-perceived pageview response time of internet services. *IEEE Trans. Parallel Distrib. Syst. 22*, 5 (2011), 773–785.