Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads

James Mickens
Microsoft Research
mickens@microsoft.com

Abstract

A modern web page contains many objects, and fetching these objects requires many network round trips—establishing each HTTP connection requires a TCP handshake, and each HTTP request/response pair requires at least one round trip. To decrease a page's load time, designers try to minimize the number of HTTP requests needed to fetch the constituent objects. A common strategy is to inline the page's JavaScript and CSS files instead of using external links (and thus separate HTTP fetches). Unfortunately, browsers only cache externally named objects, so inlining trades fewer HTTP requests now for greater bandwidth consumption later if a user revisits a page and must refetch uncacheable files.

Our new system, called Silo, leverages JavaScript and DOM storage to reduce both the number of HTTP requests *and* the bandwidth required to construct a page. DOM storage allows a web page to maintain a key-value database on a client machine. A Silo-enabled page uses this local storage as an LBFS-style chunkstore. When a browser requests a Silo-enabled page, the server returns a small JavaScript shim which sends the ids of locally available chunks to the server. The server responds with a list of the chunks in the *inlined* page, and the raw data for chunks missing on the client. Like standard inlining, Silo reduces the number of HTTP requests; however, it facilitates finer-grained caching, since each chunk corresponds to a small piece of JavaScript or CSS. The client-side portion of Silo is written in standard JavaScript, so it runs on unmodified browsers and does not require users to install special plug-ins.

1 Introduction

Users avoid slow web sites and flock towards fast ones. A recent study found that users expect a page to load in two seconds or less, and 40% of users will wait for no more than three seconds before leaving a site [1]. Thus, fast-loading pages result in happier users, longer visit times, and higher revenues for page owners. For example, when the e-commerce site Shopzilla reduced its average load time from 5 seconds to 1.5 seconds, it boosted page views by 25% and revenue by 10% [4]. Faster loads also lead to more advertising impact, since Google's AdWords system preferentially displays ads whose target pages load quickly [9]. Search engines also make loading speed a factor in their page rankings [22].

Given all of this, web designers have accumulated a series of techniques for decreasing load times. Souder's influential book *High Performance Web Sites* lists 14 of these techniques [23], with the most important one being to minimize the number of HTTP requests needed to construct a page. Over 80% of user-perceived load time is spent downloading HTML, JavaScript, images, and other objects, and 40–60% of page visitors arrive with an empty cache [25]. By minimizing the number of HTTP requests needed to build a page, developers reduce the number of round-trips needed to fetch the page's objects, and they avoid TCP slow starts for now-superfluous HTTP connections.

1.1 The Costs and Benefits of Inlining

An obvious way to eliminate HTTP requests is to make pages contain fewer objects. Unfortunately, this approach results in less interesting pages, so developers instead use *object inlining* [23]. Multiple JavaScript files are concatenated to form a smaller number of JavaScript files; similarly, multiple style sheets are combined into a single CSS file. In both cases, the number of HTTP requests decreases.

The greatest savings occur when *all* of the JavaScript and CSS is directly inserted into the page's HTML. Such aggressive inlining delivers all of the page's HTML, CSS, and JavaScript in a single HTTP fetch. Unfortunately, the significant reduction in load time comes with a price—since the browser cache can only store URL-addressable objects, the individual HTML, CSS, and

JavaScript files cannot be cached. Instead, the browser caches the aggregate inlined HTML, and if any of the embedded objects change, the browser must refetch the bytes for *all* of the objects. Ideally, we would like the best of both worlds: aggressive inlining which maintains the cacheability of the constituent objects.

1.2 Our Solution: Silo

Silo is our new framework for deploying fast-loading web applications. Silo exploits JavaScript to implement a delta-encoding HTTP protocol between an unmodified web browser and a Silo-aware web server. A Silo web server aggressively inlines JavaScript and CSS, and breaks the inlined HTML into chunks using Rabin fingerprints [18]. When a browser requests the page, the server does not return the inlined HTML—instead, it returns a small JavaScript shim which uses the browser's DOM storage [26] as a chunk cache. The shim informs the server of locally available chunks. The server responds with a list of chunk ids in the page, as well as the raw data for any chunks that do not reside on the client.

By aggressively inlining, browsers can fetch the HTML, CSS, and JavaScript for a Silo-enabled page in at most two HTTP round trips (§3.1). However, using chunking, Silo restores the cacheability that was previously destroyed by aggressive inlining. Indeed, Silo introduces a *finer* granularity of caching, since data is cached at the level of 2 KB chunks instead of entire files. This reduces bandwidth requirements when updating already cached HTML, JavaScript, and CSS files that have changed, but that retain some of their old content. Since client-side chunk data is associated with an entire domain, chunks downloaded from one page in a domain can be used to reconstruct a sibling. Thus, Silo can exploit the fact that different pages in the same domain often share content [5, 21].

1.3 Our Contributions

This paper makes the following contributions:

- We show how unmodified browsers can exploit JavaScript and DOM storage to implement a delta-encoding protocol atop standard HTTP.
- We provide new empirical data on the composition of web pages and how their content changes over time.
- We demonstrate that for pages with significant amounts of JavaScript and CSS, Silo's new protocol can reduce load times by 20%–80% while providing finer-grained caching than the standard browser cache.

We also discuss the fundamental challenge of defining "load time" in the context of modern web pages which contain rich interactive content and thousands of lines of JavaScript, Flash, and other code.

The rest of this paper is organized as follows. In Section 2, we provide background information on the HTTP protocol and describe the basic JavaScript features that Silo leverages. In Section 3, we describe how Silo uses these features to layer a custom delta-encoding protocol atop standard HTTP. Section 4 provides our PlanetLab evaluation, wherein we serve real web data under realistic network conditions to explore Silo's benefits. We discuss related work in Section 5 before concluding in Section 6.

2 Background

In this section, we provide a brief overview of the HTTP protocol, describing the particular elements that are relevant to the Silo architecture. We also describe the JavaScript features that we use to implement the client-side portion of the Silo protocol. Finally, we explain how Rabin fingerprints can be used for delta-encoding.

2.1 The HTTP Protocol

A browser uses the HTTP protocol [6] to fetch objects from a web server. A top-level page like `www.cnn.com` is composed of multiple objects. Silo separates these objects into four classes.

- HTML describes a page's content.
- Cascading style sheets (CSS) define how that content is presented.
- JavaScript code allows the page to respond to user inputs and dynamically update itself.
- Multimedia files like images, movies, and sound files provide visual and audio data.

The standard browser cache can store each class of object. However, the first three object types consist of structured, text-based data that changes relatively slowly across object revisions. In contrast, multimedia files have binary data that rarely changes in-place. Thus, only HTML, CSS, and JavaScript are amenable to delta-encoding (§2.3), a technique for describing different versions of an object with respect to a reference version. Importantly, the standard browser cache stores *whole objects* at the *URL-level*—HTTP provides no way to delta-encode arbitrary objects with respect to previously cached versions.

To increase fetch parallelism, a browser tries to open multiple connections to a single web server. HTTP is a TCP-based protocol, and TCP setup and teardown are expensive in terms of RTTs. Thus, HTTP version 1.1 introduced persistent HTTP connections, which allow a single TCP session to be used for multiple HTTP requests

and responses. On highly loaded web servers or proxies, maintaining too many persistent connections can exhaust memory, file descriptors, and other computational resources, hurting parallelism if many persistent connections are idle but no more can be created to handle new, active clients. Mindful of this threat, some proxies and web servers use stringent timeouts for persistent connections, or close them after a few objects have been transferred [2, 11].

Figure 1(a) demonstrates how a browser might download a simple web page consisting of a single HTML file and four external objects. First, the browser opens a persistent HTTP connection to the web server and fetches the page's HTML. As the browser parses the HTML, it finds references to the page's external objects. It fetches `a.css` and `c.css` using its preexisting HTTP connection; in parallel, the browser opens a second persistent connection to get `b.css` and `d.js`. The browser constructs the entire page in approximately three HTTP round trips (one RTT to fetch the HTML, and two RTTs to fetch the four objects over two persistent HTTP connections) [1].

In popular browsers like Firefox 3.0, IE 7, and Safari 3, a JavaScript fetch prevents the initiation of new parallel downloads. This is because the fetched JavaScript may change the content of the subsequent HTML (and thus the external objects that need to be fetched). Newer browsers use speculative parsing techniques to contain the side effects of erroneously fetched objects. Developers have also invented various application-level hacks to trick browsers into doing parallel JavaScript fetches [24]. Regardless, a browser can only open a finite number of parallel connections, so the fetching of non-inlined JavaScript generally adds to the load time of a page.

2.2 JavaScript

JavaScript [7] is the most popular language for client-side scripting in web browsers. With respect to Silo, JavaScript has three salient features. First, JavaScript programs can dynamically modify the content of a web page. Second, JavaScript can associate large amounts of persistent local data with each web domain. Third, JavaScript can use AJAX calls [7] to construct new communication protocols atop HTTP.

2.2.1 Manipulating the DOM

JavaScript represents the state of a web page using the Document Object Model (DOM) [28]. The DOM provides a standard, browser-neutral API for querying and manipulating the presentation and content of a page. In the context of Silo, the most important DOM calls are the ones which allow pages to overwrite their own content.

- When a JavaScript program calls `document.open()`, the browser clears any preexisting presentation-layer data associated with the page, i.e., the JavaScript state is preserved, but the page's old HTML is discarded.
- The application writes new HTML to the page using one or more calls to `document.write(html_str)`.
- Once the application has written all of the new HTML, it calls `document.close()`. This method instructs the browser to finish parsing the HTML stream and update the presentation layer.

Using these calls, a web page can completely overwrite its content. Silo leverages this ability to dynamically construct pages from locally cached data chunks and new chunks sent by Silo web servers.

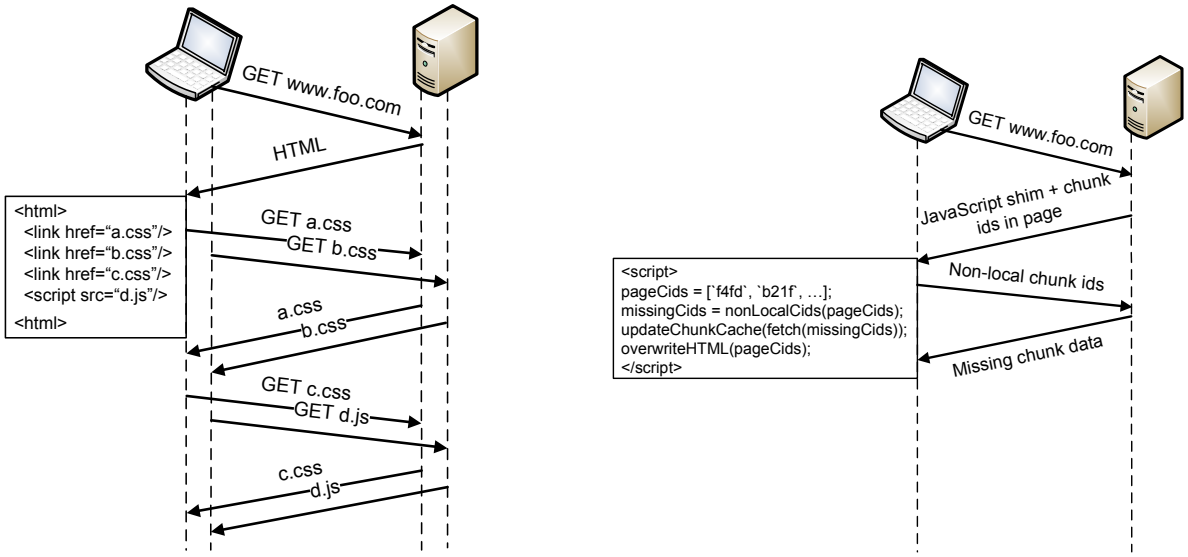
2.2.2 Associating Web Domains With Local Data

JavaScript does not provide an explicit interface to the browser cache. JavaScript-generated requests for standard web objects like images may or may not cause the associated data to lodge in the cache, and JavaScript programs cannot explicitly write data to the cache. Furthermore, there is no way for a JavaScript program to list the contents of the cache.

For many years, the only way for JavaScript to store persistent, programmatically-accessible client-side data was through cookies [10]. A cookie is a small file associated with a particular web domain. When the user visits a page belonging to that domain, the browser sends the cookie in the HTTP request. The domain can then read the cookie and send a modified version in the HTTP response. JavaScript provides a full read/write interface for cookies. However, browsers restrict the size of each cookie to a few kilobytes, making cookies unsuitable for use as a full-fledged data store.

To solve this problem, Google introduced Gears [8], a browser plugin that (among other things) provides a SQL interface to a local data store. Although Gears provides a powerful storage API, it requires users to modify their browsers. Luckily, modern browsers like IE8 and Firefox 3.5 support a new abstraction called DOM storage [26]. DOM storage allows a web domain to store client-side key/value pairs. By default, browsers allocate 5–10 MB of DOM storage to each domain. The DOM storage API has been accepted by the W3C Web Apps Working group [26] and will likely appear in the upcoming HTML5 standard. Given this fact, Google recently announced that it was ramping down active development on Gears, and that it expected developers to

[1] HTTP 1.1 allows clients to pipeline multiple requests over a single connection, but many web servers and proxies do not support this feature or support it buggily. Pipelining is disabled by default in major browsers.



(a) The standard HTTP protocol.

(b) The basic Silo protocol. See Section 3.2 for optimizations.

Figure 1: Fetching a simple web page.

migrate to the standardized HTML5 storage mechanisms once HTML5 gained traction [14].

Web applications most commonly use DOM storage to buffer updates during disconnected operation. Silo uses DOM storage in a much different fashion, namely, as an application-level chunk cache that enables delta-encoding (§2.3) for web pages.

2.2.3 AJAX

A JavaScript program can use the AJAX interface [7] to explicitly fetch new web data. AJAX data is named by URL, and a JavaScript application can inspect and set the headers and the body of the HTTP request and response. Thus, an application can use AJAX to layer arbitrary client-server protocols atop HTTP. Silo uses AJAX to implement a custom delta-encoding HTTP protocol inside unmodified web browsers.

2.3 Delta-encoding

Delta-encoding is a technique for efficiently describing the evolution of a data object. Each version of the object is represented as a set of edits or "deltas" applied to a reference version of the object. Once a client downloads the full reference object, it can cheaply reconstruct a newer version by downloading the deltas instead of the entire new object.

Many distributed systems employ *chunk-based* delta-encoding. Each object is broken into small, contiguous byte ranges; a single edit modifies one or more of these chunks. Hosts transmit their edits by sending the positions of deleted chunks, and the positions and data of new chunks.

If chunk boundaries are determined by fixed byte offsets, an edit which increases or decreases an object's size will invalidate all chunks after the edit point. To avoid this problem, distributed systems typically eschew fixed length chunks and use *content-delimited* chunks instead. In these systems, chunk boundaries are induced by special byte sequences in the data. LBFS [16] popularized a chunking method in which applications push a sliding window across a data object and declare a chunk boundary if the Rabin hash value [18] of the window has N zeroes in the lower-order bits. By varying N, applications control the expected chunk size. With content-based hashing, an edit may create or delete several chunks, but it will not cause an unbounded number of chunk invalidations throughout the object.

Distributed systems typically name each chunk by the SHA1 hash of its content. This allows different hosts to independently pick the same name for the same chunk. Hosts can determine whether they store the same version of an object by exchanging the chunk ids in their copies of the file. By comparing these lists, a peer can determine whether it needs to delete content from its local version or fetch new chunks from the other host. Silo uses a similar protocol to delta-encode the transmission of previously viewed web pages.

3 Silo Architecture

Ideally, the Silo protocol would be implemented as an extension to HTTP, and commodity web servers and browsers would ship with native Silo support. However, to ease deployability, our current Silo architecture lever-

ages JavaScript to execute on *unmodified* browsers. Web servers must still be modified, but this is much less onerous than modifying millions of end-user browsers.

Our Silo architecture consists of three components: a Silo-aware web server, an unmodified client browser, and a JavaScript shim that is generated by the server and which implements the client side of the Silo protocol. In this section, we describe this architecture in more detail. We also describe several optimizations to the basic Silo protocol; some of these optimizations mask performance issues in current JavaScript engines, and others leverage cookies to reduce the number of RTTs needed to construct a page.

Silo's goal is to reduce the time needed to assemble a page's HTML, CSS, and JavaScript. Borrowing Firefox's event terminology, we refer to this time as the page's `DOMContentLoaded` time [15]. Fetching a page's HTML, CSS, and JavaScript is necessary but often insufficient to produce a fully functioning page. For example, pages often contain multimedia files which are not amenable to Silo-style delta encoding. Silo is orthogonal to techniques for improving the load times of these objects. We return to this topic when we describe our evaluation methodology (§4.1).

3.1 The Basic Protocol

Figure 1(a) depicts how a web page is constructed using the standard HTTP 1.1 protocol. The browser first retrieves the HTML for the page. As it parses the file, it issues parallel fetches for the externally referenced objects. In Figure 1(a), we assume that the client cache is empty, and that the browser can issue two fetches in parallel. Thus, the browser must use three HTTP round trips to construct the page (one to send the initial `GET`, and two to download the external objects).

Figure 1(b) depicts how a Silo-enabled page is fetched. The browser issues a standard `GET` for the page, but the web server does not respond with the page's HTML. Instead, the server sends a small piece of JavaScript which acts as the client-side participant in the Silo protocol. The JavaScript shim contains an array called `pageCids`; this array lists the ids of the chunks in the page to construct. The shim inspects the client's DOM storage to determine which of these chunks do not reside locally. The shim uses a synchronous AJAX POST to send the missing chunk ids to the server. The server replies with the raw data for the missing chunks. The client assembles the relevant chunks and overwrites the page's current HTML, reconstructing the original inlined page. In this fashion, the basic Silo protocol uses two HTTP round trips to fetch an arbitrary number of HTML, CSS, and JavaScript files.

```
<html>
   <script>
      /*Code for Silo shim*/
   </script>
   <style type=text/css>
      /*Inlined css for a.css*/
   </style>
   ...
   <script>
      /*Inlined JavaScript for d.js*/
   </script>
</html>
<!--- Chunk manifest
cid0, offset0, len0
cid1, offset1, len1
...
--->
```

Figure 2: When the client chunk cache is empty, the Silo server responds with inlined HTML which is immediately usable by the browser. The client shim asynchronously parses the chunk manifest at the bottom of the HTML and updates the local chunk cache.

3.2 Optimizations

Handling Cold Client Caches: At any given moment, 40%–60% of the users who visit a page will have no cached data for that page [25]. However, as shown in Figure 1(b), a Silo server does not differentiate between clients with warm caches and clients with cold caches—in either case, the server's second message to the client is a string containing raw data for N chunks. If the client has an empty cache, it must *synchronously* perform N substring operations before it can extract the N chunks and recreate the inline page. In current browsers, this parsing overhead may be hundreds of milliseconds if inlined pages contain hundreds of KB of data (and thus hundreds of chunks).

To improve load times in these situations, Silo sends a different second message to clients with empty chunk caches. Instead of sending raw chunk data that the client must parse before it can reconstruct the page, the server sends an inlined version of the page annotated with a special *chunk manifest* at the end (see Figure 2). The chunk manifest resides within an HTML comment, so the client can commit the annotated HTML immediately, i.e., without synchronously performing substring operations. Later, the client asynchronously parses the manifest, which describes the chunks in the inlined HTML using a straightforward offset+length notation. As the client parses the manifest, it extracts the relevant chunks and updates the local chunk cache.

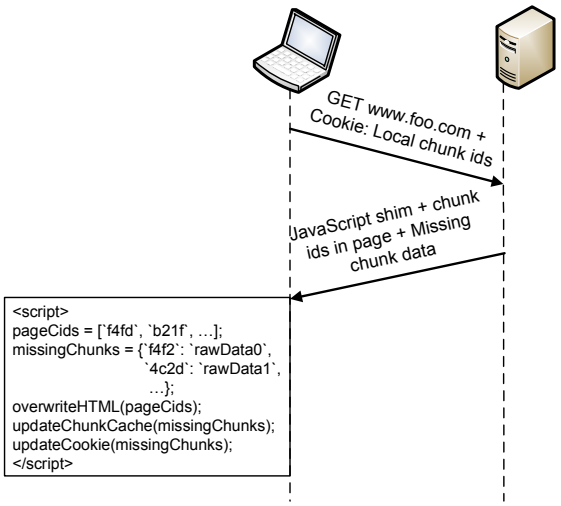


Figure 3: Single RTT Silo protocol, warm client cache.

Leveraging HTTP Cookies: Silo uses cookies (§2.2.2) in two ways to reduce the number of HTTP round trips needed to construct a page. First, suppose that a client has an empty chunk cache. Even if the server uses annotated HTML to eliminate synchronous client-side string operations, the client must expend two HTTP round trips to construct the page. However, the client shim can set a "warm cache" variable in the page's cookie whenever it stores chunks for that page. If the server does not see this cookie variable when the client sends the initial HTTP `GET` operation, the server knows that the client chunk cache is cold, either because the client has never visited the page before (and thus there is no cookie), or the client has visited the page before, but the local cache is empty for some other reason (e.g., because previous writes to DOM storage failed due to a lack of free space). Regardless, the server responds to the initial HTTP `GET` with annotated HTML. This allows a Silo client with a cold cache to fetch all of the HTML, CSS, and JavaScript in a single HTTP round trip.

Clients with warm caches can also use cookies to indicate the contents of their cache. Whenever the client shim updates the chunk cache for a page, it can add the new chunk ids to the page's cookie. The next time that the browser requests the page, the server can inspect the cookie in the initial HTTP `GET` and determine which page chunks already reside on the client. The server can then directly respond with the Silo shim and the missing chunks, eliminating the second HTTP round trip required by the basic Silo protocol.

Browsers typically restrict cookie sizes to 4 KB, the minimum cap allowed by RFC 2109 [10]. SHA1 hashes are 20 bytes, so a cookie could hold a maximum of 204 SHA1 chunk ids. If chunks are 2 KB on average, then a cookie could reference a maximum of 408 KB of local chunk cache data. For many popular web pages, 408 KB is sufficient to delta-encode several versions of the page (§4.3). However, some pages are so big that even a single snapshot of their HTML, CSS, and JavaScript will not fit in 408 KB of chunk-addressable storage. Fortunately, Silo can expand the addressable range by leveraging the fact that Silo servers have complete control over chunk names. Name-by-hash allows different machines to agree on chunk names without a priori communication, but Silo clients never assign chunk ids—servers always determine the mapping between chunk ids and chunk data. Thus, servers can use ids that are much shorter than SHA1 hashes. For example, a server could assign each new, unique chunk a strictly increasing 3 byte id. Since client-side DOM storage is partitioned by domain, these ids only have to be collision-free within a domain, not the entire Internet. With 3 byte ids, a domain could define over 16 million unique chunks before having to "reset" the namespace. A 4 KB cookie could store 1365 of these 3 byte ids. With 2 KB chunks, this would allow clients to name roughly 2.7 MB of local cache data. As we show in Section 4.3, 2.7 MB of storage should be more than sufficient to delta-encode multiple versions of a page's HTML, CSS, and JavaScript.

Finally, we note that Silo is agnostic to the specific method in which data is chunked. Indeed, websites are free to pick the granularity of caching that best suits their needs. For example, if a web site knows that its HTML evolves in a structured way, it can define an HTML-specific chunking function for its pages. Alternatively, a site could decide to chunk at the whole file level, i.e., with each CSS and JavaScript file residing in its own singleton chunk. Silo is not bound to a specific chunking mechanism—it merely leverages chunking to provide inlining without destroying the cacheability of individual objects.

The Full Protocol: Given all of these optimizations, we now describe the full version of the Silo protocol. We separate our description into two cases: when the client has a cold chunk cache, and when the client has a warm cache.

- *Cold cache:* The client generates an HTTP `GET` request for a page, sending a cookie which indicates a cold cache. The server responds with annotated HTML as shown in Figure 2. The browser commits the annotated HTML immediately; asynchronously, the Silo shim parses the chunk manifest, extracts the associated chunks, and writes them to DOM storage. In this scenario, Silo needs one HTTP round trip to assemble all of the page's HTML, CSS, and JavaScript.
- *Warm client cache:* The client generates an HTTP `GET` request for a page, sending a cookie which in-

dicates a warm cache. If the client can fit all of the local chunk ids within the cookie, it can receive the Silo shim and the missing chunk data in a single server response (see Figure 3). Otherwise, it falls back on the basic, two RTT Silo protocol depicted in Figure 1(b).

As we show in the evaluation section, Silo's primary benefit is to reduce load times for clients with cold caches; client with completely warm caches have no fetch latencies to mask. However, for clients with only partially warm caches, Silo reduces the fetch penalty since an arbitrary number of stale objects can be updated using at most two HTTP round trips. Furthermore, if pages use fine-grained chunking, data can be invalidated at a much finer level, reducing the fetch bandwidth in addition to the fetch latency.

3.3 Other Design Decisions

When a client has a partially warm cache, Silo asynchronously writes new chunks to disk. As we show in Section 4.2, Firefox's writes to DOM storage can require more than a hundred milliseconds. Thus, to avoid foreground resource contention during the page load, Silo synchronously extracts in-memory versions of the new chunks needed to assemble the page, but it always defers writes to stable storage for a few seconds (our current prototype waits 5 seconds).

The regular browser cache stores data belonging to `<script>` tags. These tags can store arbitrary data as JavaScript strings. Thus, Silo could use the standard browser cache as a chunk store by writing blobs to scripts whose names were chunk ids. At first glance, this approach seems attractive since it obviates the need for a separate chunk cache, and it would work on unmodified browsers. However, browsers provide no way for JavaScript applications to explicitly insert data into the cache. Instead, applications implicitly warm the browser cache as a side-effect of fetching external data. Even simple web pages will likely contain at least a few tens of chunks; thus, a client which wanted to store these chunks using `<script>` tags would have to issue a large number of HTTP requests. This would obviously lead to huge increases in page load times. Thus, `<script>` tags are a poor substitute for DOM storage.

4 Evaluation

In this section, we evaluate Silo by serving real web content from a Silo deployment on PlanetLab. We demonstrate that Silo can substantially improve load times for pages with large amount of CSS and JavaScript. We also provide an analysis of how content chunks evolve within the same page and across different pages.

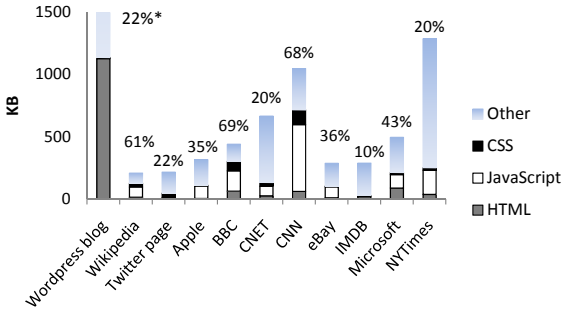
4.1 Methodology

Gathering and Serving Real Web Data: For our Silo-aware web server, we used a modified version of the Mugshot replay proxy [13]. Mugshot is a system for capturing and replaying the behavior of JavaScript-based web applications. A browser-side Mugshot JavaScript library records the bulk of the nondeterminism like GUI activity and calls to the random number generator. A special Mugshot proxy sits between the real web server and the browser; this proxy records the binding between the URLs in client HTTP requests and the data that is returned for those URLs. Later, at replay time, Mugshot uses the replay proxy as a web server, letting it respond to data requests from the replaying client code. This ensures that the data fetched at replay time is the same data that was fetched at logging time.

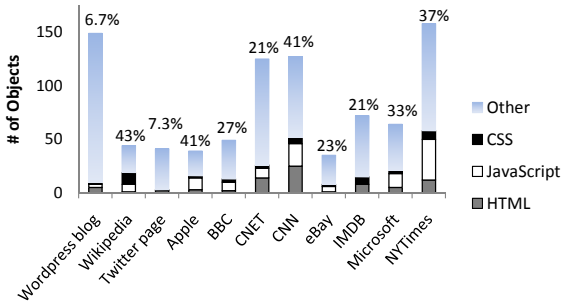
To test the Silo protocol, we first ran the Mugshot proxy in its standard logging mode, capturing HTTP headers and data from real websites. We then switched the proxy into replay mode and had clients use it as their web proxy. When clients requested a page whose content we previously logged, the proxy served that page's objects directly from its cache. We modified the replay mode to support the Silo chunking protocol described in Section 3.2. Thus, we could simulate Silo's deployment to an arbitrary web site by logging a client's visit to that site, clearing the client-side cache, and then revisiting the page in Silo-enabled replay mode.

Experimental Setup: Our experimental setup consisted of a client machine whose browser fetched content from a Silo-aware web server. The client browser resided on a Lenovo ThinkPad laptop running Windows 7. The laptop had an Intel Core 2 Duo with 2.66 GHz processors and 4GB of RAM. The server ran on a PlanetLab node with a 2.33 GHz Intel Core Duo and 4 GB of RAM. The client communicated with the server over a residential wireless network. Across all experiments, the RTT was stable at roughly 150 ms, and the bandwidth varied between 700–850 Kbps. Thus, network conditions were similar to those experienced by typical cable modems or DSL links. In the results presented below, we used the Firefox 3.5.7 browser, but the results for IE8 were similar. We present Firefox results because IE8 does not yet define the `DOMContentLoaded` event, whose usage we describe shortly. Silo using Rabin chunking with an expected chunk size of 2 KB.

In real life, when a browser loads a page, it opens multiple simultaneous connections to multiple servers and proxies. In our experimental setup, the client browser fetched everything from a single Mugshot proxy. To ensure that we did not unduly constrain fetch parallelism, we configured Firefox to open up to sixteen persistent connections to a proxy instead of the default of eight.

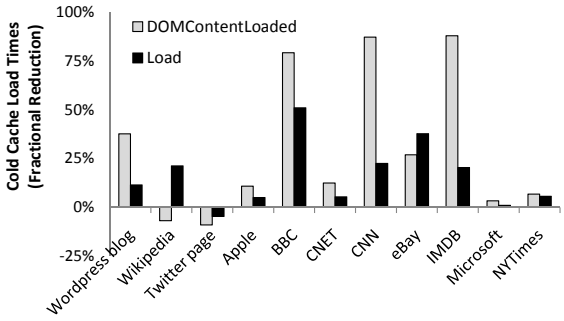


(a) Content type by byte size. ∗Wordpress blog contains 5.1 MB.

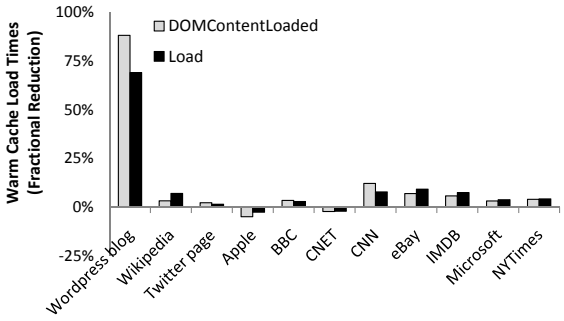


(b) Content type by fraction of objects.

Figure 4: Content statistics for several popular web pages. Top-level fractions are the percentage of content that is potentially inlineable (HTML+CSS+JavaScript).



(a) Cold cache.



(b) Warm cache.

Figure 5: Page load times.

In an actual Silo deployment, a web server would always compress the chunk manifests that it sent to clients. However, we found that many web servers do not compress the HTML and the CSS that they transmit [2]. To provide a fairer comparison, the Silo server only `gzip`-ed its chunk manifest if the logged HTML page was also compressed. Our Silo server also closed persistent connections whenever the logged persistent connection was closed.

Defining Load Time: One of Silo's benefits is that it reduces page load time. The intuition is straightforward—by decreasing the number of round trips needed to construct a page, the browser spends less time stalling on the network and builds pages faster. Unfortunately, providing a precise definition of load time is difficult.

At the end of the exchanges shown in Figures 1(a) and (b), the page's HTML, JavaScript, and CSS have been fetched and parsed. The browser has calculated the layout and presentation format of the content, but it may not have fetched some of this content. In particular, multimedia objects like images and advertisements may or may not have been downloaded in parallel with the DOM content. Thus, the "full" page load may not coincide with the completion of the DOM load.

To further complicate matters, some sophisticated web pages use JavaScript code to defer certain fetches. For example, upon initial load, some sites defer the fetching of content at the (not yet scrolled-to) bottom of the page. Some pages also split their fetches into a lightweight "top-half" which quickly displays visual elements, and a more heavyweight "bottom-half" which grabs the large resources that are actually represented by the GUI elements. Defining load times in these cases is difficult—the initial page load may seem quick, but if users try to access the rich data too quickly, they may experience a second, more painful load time.

Firefox issues the `DOMContentLoaded` JavaScript event when it has fetched all of a page's HTML, CSS, and JavaScript. It fires the `load` event when *all* of the page's content has been fetched. Silo definitely reduces the time to `DOMContentLoaded`; in the simple example of Figure 1, this time is reduced from three RTTs to two. Silo typically reduces the time to `load` as well. However, for any given page, `load` time is usually heavy-tailed [17, 19, 27]. This is caused by a variety of events, such as random network congestion which slashes throughput for some TCP connections, or heav-

[2] JavaScript is rarely sent compressed since several browsers have buggy decompression routines for scripts.

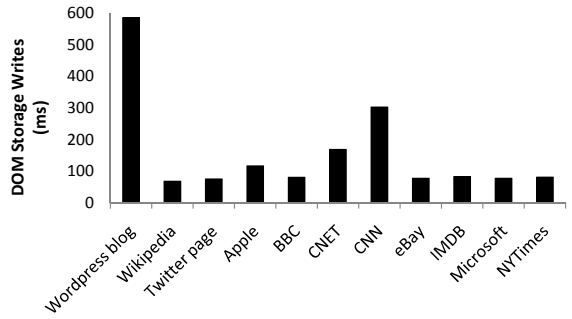


Figure 6: DOM storage write overheads.

ily loaded servers for widely shared third-party images or advertisements. Techniques for optimizing load times for multimedia files, e.g., through image spriting [23], are complimentary but orthogonal to Silo. However, for the sake of completeness, our results in Section 4.2 describe Silo's impact on both `DOMContentLoaded` time and `load` time.

4.2 Reducing Page Load Times

Figure 4 shows the content statistics for several popular web pages. It is interesting to note the size of the JavaScript files; as shown in Figure 4(a), many websites have hundreds of KB of JavaScript code, and JavaScript makes up more than 60% of the byte content for the front pages of Wikipedia, BBC, and CNN. This result is perhaps counterintuitive, since modern web applications are popularly characterized as being multimedia-heavy. This conventional wisdom is certainly true, but it does underestimate the pervasiveness and the size of JavaScript code.

Whereas Figure 4(a) breaks down content type by byte size, Figure 4(b) describes content type by object count. Viewed from this perspective, there are fewer opportunities for inlining. For example, in the popular DavisW Wordpress blog, 22% of all bytes are HTML, but only 6.7% of distinct objects are HTML. Similarly, in the BBC front page, 69% of all bytes belong to HTML, JavaScript, and CSS, but only 27% of the total objects are inlineable. The number of distinct objects governs the number of HTTP requests needed to build a page. Thus, the difference between Figure 4(a) and (b) may seem to doom any efforts to reduce load times through inlining. However, web designers typically structure pages such that the most important objects load first. For example, code for ad-generation may load asynchronously, and embedded movie players often do not start to prefetch data until the rest of the page has loaded. Thus, Silo still has many opportunities to reduce load times.

Figure 5 shows how quickly Silo-enabled pages load in comparison to their "standard" versions. Silo's benefit is measured as the fraction of the standard load time that Silo eliminates. The best achievable reduction is 100%, and negative percentages are possible if a Silo page loads slower than its standard counterpart. Figure 5(a) depicts Silo's performance when client caches are empty, i.e., we compare a Silo page load with an empty chunk cache to a load of the regular page when the standard browser cache is empty. In five of the eleven websites, Silo reduces `DOMContentLoaded` times by 25% or more. The other sites have fewer synchronous object fetches on the critical path for page loading, so Silo-enabled versions of these pages load no faster, or even slightly slower due to Silo's computational overheads. However, Silo generally does little harm, and often provides non-trivial benefits.

Unsurprisingly, Figure 5(a) shows that Silo reduces `DOMContentLoaded` times more than it reduces `load` times. However, we emphasize that the `DOMContentLoaded` event represents the earliest point at which a page is ready for human interaction, so minimizing `DOMContentLoaded` time is worthwhile. We also note that for five of the eleven sites, Silo also reduces `load` times by 20% or more.

Figure 5(b) shows load times when clients have warm caches and Silo uses the single RTT protocol described in Figure 3. In all cases, caches were warmed with data from 9 AM on May 11, 2010, and browsers were subsequently directed towards replayed page content from an hour later. Silo generally provides few benefits when caches are warm—few (if any) new objects must be fetched, so there are fewer network round trips for Silo to mask. However, Silo did provide a large benefit to the Wordpress site, since the small change in HTML content necessitated the transfer of the entire 1 MB file in the standard case, but only a few chunks in Silo's case. In this scenario, Silo reduced latency not by hiding a round trip, but by dramatically reducing the amount of data that had to be shipped across the wide area.

Silo reads from DOM storage to gather locally cached chunks, and it writes to DOM storage to cache new blocks sent by the server. In Firefox's current implementation of DOM storage, even small reads and writes are fairly expensive, with writes being two to five times slower than reads. Figure 6 shows Silo's write throughput for committing all of a page's chunks, showing that this operation, undertaken when clients have completely cold caches, generally requires 50–150 ms. The DOM storage API is a new browser feature that is currently unused by most sites, but we expect its throughput to improve as the API becomes more popular and browser implementers have a stronger motivation to make it fast.

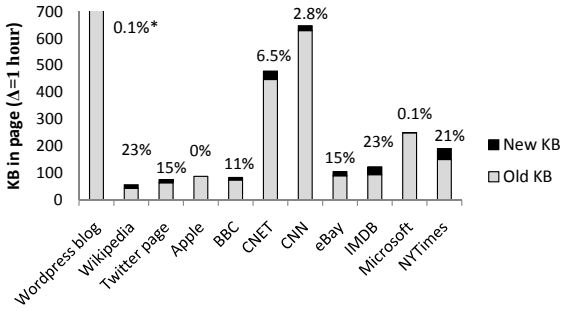


Figure 7: Byte turnover: 1 hour.

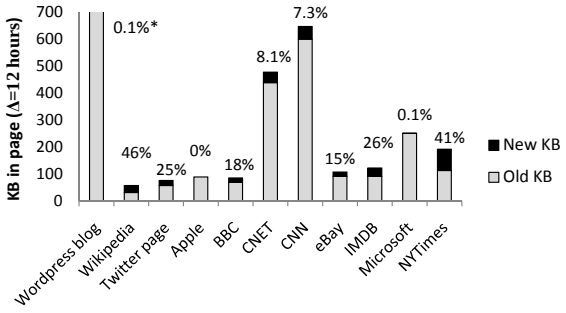


Figure 8: Byte turnover: 12 hours.

4.3 Turnover Rates for Inlineable Data

Figures 7, 8, and 9 show how inlineable page content evolved over a 24 hour period starting at 9:30 AM on Friday, January 8, 2010. These results show that byte turnover varies widely across pages. For example, the Apple website showed no variation at all during this particular period. The Wordpress blog also showed little change, since the bulk of its content consisted of a large HTML file with minimal deltas and a set of static CSS. In contrast, the New York Times site had 21% of its chunks replaced in an hour; after a day, almost half of the chunks were new. The Wikipedia front page had a similar level of turnover, since it also rotates top-level stories frequently. Interestingly, despite it high level of visual turnover, CNN only generated 2.8% new chunks in an hour, and 7.6% new chunks over the period of a day. This is because CNN contained large amounts of stable JavaScript and CSS (see Figure 4).

During the observation period, most byte turnover resulted from changes to a page's HTML. However, CNET, CNN, and the New York Times occasionally added or deleted JavaScript files which managed advertisements. CNN also replaced a 522 byte chunk in a CSS file.

Figure 10 depicts the level of byte sharing between different pages in the same domain. We distinguish between a top-level front page, e.g., `www.cnn.com`, and a second-level page which is directly referenced by the front page. Figure 10(a) depicts the average similarity

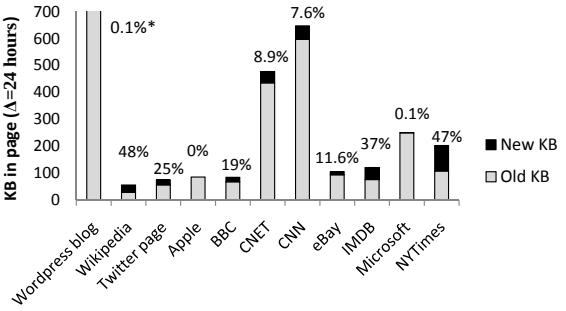


Figure 9: Byte turnover: 24 hours.

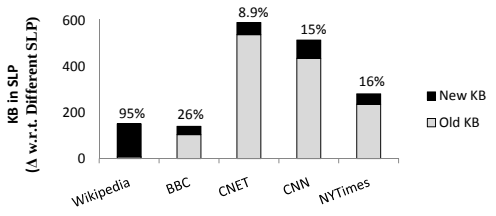
of five random second-level pages to each other, whereas Figure 10(b) shows the average similarity of these pages to the front page. These results show that second-level pages have more in common with each other than with their front pages. However, the CNN and CNET front pages offer significant opportunities for clients to warm their chunk caches for subsequent second-level page views.

When examining the second-level New York Times pages, Silo reported a large amount of redundant data. Upon checking Silo's log files, we discovered that each page repeated the same `<script>` tag four times. The script was a piece of advertising management code hosted by a third party. Its multiple inclusions were apparently harmless in terms of the page's behavior, but scripts are obviously not required to be idempotent. Thus, Silo's chunking scheme is useful for alerting content providers to potentially unintended duplication of scripts.

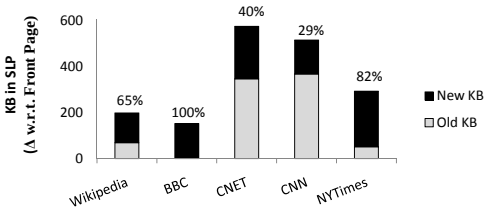
5 Related Work

Silo's most direct inspiration was LBFS [16], a network file system layered atop a distributed chunk store. In LBFS, each file is represented as a set of Rabin-delimited chunks. Clients and servers maintain an index of locally stored chunks. Whenever hosts must exchange files, they transfer chunk manifests indicating the data involved in local file operations; a host only fetches raw chunk data if no local copy exists.

Silo uses the advanced programming environment of modern browsers to implement an LBFS-like protocol atop HTTP. Value-based Web Caching [20] has a similar goal. VBWC introduces two new proxies, one inside the network belonging to an ISP, and another on the client. The browser issues web requests through the local proxy, and the local proxy engages in an LBFS protocol with the ISP proxy, which fetches and chunks web content. Unlike Silo, VBWC requires modification to client machines, hindering deployability. VBWC also does not



(a) Similarity of two second-level pages.



(b) Similarity of front page and second-level page.

Figure 10: Intra-domain page similarity. Top-of-bar fractions are the percentages of new content.

exploit object inlining to reduce the volume of HTTP requests issued by the client.

A variety of other projects have explored delta-encoding for web traffic. For example, Douglis studied the degree of similarity between different pages on the same site [5]. Chan described how objects that already reside in a browser cache can act as reference objects for delta-encoding new files [3]. Savant extended both lines of research, showing that delta-encoding of HTML files from the same site can achieve compression ratios of greater than 80% [21].

Web developers can reduce user-perceived load times by deferring the fetches for components which are not immediately needed. For example, Yahoo ImageLoader [29] provides a JavaScript framework for delaying the load of images that the user will not need within the first few seconds of viewing the page; such images might be positioned beneath the initially visible portion of the page, or they might only be needed if the user performs a certain action. The Doloto [12] tool provides a similar service for JavaScript code. Doloto analyzes a pages JavaScript behavior and identifies two sets of code: code which is invoked immediately, and code which is used infrequently, or only used a few seconds after the initial page load. After collecting this workload data, Doloto rewrites the application code, loading the former code set at page initialization time, and lazily downloading the latter set, only fetching it on demand if its functionality is actually needed. Silo is orthogonal to projects like ImageLoader since its delta-encoding does not apply to multimedia files. Silo is complimentary to projects like Doloto since Silo can reduce the transfer time of any JavaScript that Doloto labels as "immediately necessary."

6 Conclusions

Slow web pages frustrate users and decrease revenues for content providers. Developers have created various ways to defer or hide fetch latencies, but perhaps the most effective technique is the most straightforward: reducing the number of HTTP requests required to build a page. Unfortunately, this strategy presents content providers with a quandary. They can reduce the number of objects in each page, but this can negatively impact the rich content of the page. Alternatively, the content provider can inline the bodies of previously external JavaScript and CSS files. However, this destroys the cacheability of these files, since standard browser caches only store objects named via external URL pointers.

In this paper, we introduce Silo, a new framework for reducing load times while preserving cacheability. Silo exploits JavaScript and DOM storage to implement a delta-encoding protocol atop standard HTTP. Using the Silo protocol, a web server can aggressively inline JavaScript and CSS without fear of losing cacheability. Indeed, since Silo has complete control over its DOM storage cache, it can provide a *finer* granularity of caching than that provided by the browser. Silo's client-side component consists of standard JavaScript, meaning that Silo can be deployed to unmodified browsers. Experiments show that Silo's inlining and chunking protocol can reduce load times by 20%–80% for pages with large amounts of JavaScript and CSS. Additionally, a Silo web server's chunking facilities, in concert with its ability to record HTTP sessions, provide a useful platform for studying the turnover rate of data in web pages.

References

[1] AKAMAI TECHNOLOGIES. Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times, September 14 2009. `http://www.akamai.com/html/about/press/releases/2009/press_091408.html`.

[2] CHAKRAVORTY, R., BANERJEE, S., CHESTERFIELD, J., RODRIGUEZ, P., AND PRATT, I. Performance Optimizations for Wireless Wide-area Networks: Comparative Study and Experimental Evaluation. In *Proceedings of Mobicom* (Philadelphia, PA, September 2004), pp. 159–173.

[3] CHAN, M. C., AND WOO, T. Cache-based Compaction: A New Technique for Optimizing Web Transfer. In *Proceedings of INFOCOM* (New York, NY, March 1999), pp. 117–125.

[4] DIXON, P. Shopzilla Site Redesign: We get what we measure. In *Presentation at O'Reilly Conference on Web Performance and Operations* (June 2009).

[5] DOUGLIS, F., AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of USENIX Technical* (San Antonio, TX, June 2003), pp. 113–126.

[6] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[7] FLANAGAN, D. *JavaScript: The Definitive Guide*, 5 ed. O'Reilly Media, Inc., 2006.

[8] GOOGLE. Gears: Improving Your Web Browser. `http://gears.google.com/`, 2008.

[9] GOOGLE. AdWords: How does load time affect my landing page quality? `http://adwords.google.com/support/aw/bin/answer.py?answer=87144`, 2009.

[10] KRISTOL, D., AND MONTULLI, L. HTTP State Management Mechanism. RFC 2109 (Draft Standard), February 1997.

[11] LIN, X.-Z., WU, H.-Y., ZHU, J.-J., AND WANG, Y.-X. On the Performance of Persistent Connection in Modern Web Servers. In *Proceedings of the ACM Symposium on Applied Computing* (Fortaleza, Ceara, Brazil, March 2008), pp. 2403–2408.

[12] LIVSHITS, B., AND KICIMAN, E. Doloto: Code Splitting for Network-Bound Web 20 Applications. In *Proceedings of SIGSOFT Symposium on the Foundations of Software Engineering* (2008), pp. 350–360.

[13] MICKENS, J., HOWELL, J., AND ELSON, J. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI* (San Jose, CA, April 2010).

[14] MILIAN, M. What's powering Web apps: Google waving goodbye to Gears, hello to HTML5. In *Los Angeles Times* (November 30 2009). `http://latimesblogs.latimes.com/technology/2009/11/google-gears.html`.

[15] MOZILLA DEVELOPER CENTER. Gecko-Specific DOM Events. `https://developer.mozilla.org/en/Gecko-Specific_DOM_Events`.

[16] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A Low-bandwidth Network File System. In *Proceedings of SOSP* (Banff, Canada, October 2001), pp. 174–187.

[17] OLSHEFSKI, D., NIEH, J., AND AGRAWAL, D. Inferring Client Response Time at the Web Server. In *Proceedings of SIGMETRICS* (Marina del Rey, CA, June 2002), pp. 160–171.

[18] RABIN, M. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[19] RAJAMONY, R., AND ELNOZAHY, M. Measuring Client-Perceived Response Times on the WWW. In *Proceedings of USITS* (San Francisco, CA, March 2001).

[20] RHEA, S., LIANG, K., AND BREWER, E. Value-Based Web Caching. In *Proceedings of the World Wide Web Conference* (Budapest, Hungary, May 2003), pp. 619–628.

[21] SAVANT, A., AND SUEL, T. Server-Friendly Delta Compression for Efficient Web Access. In *Proceedings of the International Workshop on Web Content Caching and Distribution* (Hawthorne, NY, September 2003).

[22] SINGHAL, A., AND CUTTS, M. Using site speed in web search ranking. `http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html`, April 9 2010.

[23] SOUDERS, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, Cambridge, MA, 2007.

[24] SOUDERS, S. Loading Scripts Without Blocking. In *High Performance Web Sites blog* (April 27 2010). `www.stevesouders.com/blog/2009/04/27/loading-scripts-without-blocking`.

[25] THEURER, T. Performance Research, Part 2: Browser Cache Usage–Exposed! In *Yahoo User Interface Blog* (January 4 2007). `yuiblog.com/blog/2007/01/04/performance-research-part-2`.

[26] W3C WEB APPS WORKING GROUP. Web Storage: W3C Working Draft. `http://www.w3.org/TR/2009/WD-webstorage-20091029`, October 29 2009.

[27] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well–Conditioned, Scalable Internet Services. In *Proceedings of SOSP* (Chateau Lake Louise, Canada, October 2001), pp. 230–243.

[28] WORLD WIDE WEB CONSORTIUM. Document Object Model. `http://www.w3.org/DOM`, 2005.

[29] YAHOO! YUI 2: ImageLoader. `http://developer.yahoo.com/yui/imageloader`, 2010.